(1)

SUBMICRON SYSTEMS ARCHITECTURE

SEMIANNUAL TECHNICAL REPORT

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

5103:TR:83

Computer Science
California Institute of Technology

November 1983

SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

California Institute of Technology

November 1983

5103:TR:83

Reporting Period: 16 April 1983 to 15 October 1983

Principal Investigator: Charles L Seitz

Faculty Investigators: Randal E Bryant
James T Kajiya
Alain J Martin
Robert J McEliece
Martin Rem
Charles L Seitz

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

SUBMICRON SYSTEMS ARCHITECTURE

Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document reports the research activities and results for the period 16 April 1983 to 15 October 1983 under the Defense Advanced Research Project Agency (ARPA) Submicron Systems Architecture Project.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes, and includes related efforts in concurrent computation and VLSI design. Additional background information can be found in previous semiannual technical reports [5052:TR:82, 5078:TR:83].

1.3 Highlights

The highlights of the previous 6 months are:

(1) The cosmic cube, a 64-element experimental homogeneous machine, was completed and is now in regular use. Benchmarks of this system show it outrunning a VAX11/780 by a factor of 6 on two large regular computations. Numerous other application programs are in progress for this machine. A new operating system, the cosmic kernel, has been defined and its code is being written.

(2) Prototype mosaic processors have been packaged with fast off-chip storage for a small mosaic-tree for software experiments. Meanwhile, the efforts in the design of the single chip mosaic element are approaching completion, with the storage section designed and several processor improvements accomplished, including interrupts, a multiply intruction, a faster control PLA, and new microcode.

(3) The algorithms for and logical design of the super mesh element are complete and have been simulated with MOSSIM.

(4) FMOSSIM, a concurrent fault simulator for MOS digital systems, is now operational. This program uses the same switch-level representation of MOS circuits as the logic simulator MOSSIM II, and so can model such MOS circuit structures as (bidirectional) pass transistors, static and precharged logic, busses, and both static and dynamic memory. The concurrent simulation techniques of FMOSSIM simultaneously models the good circuit and a large number of faulty circuits, and consequently requires much less CPU time than simple serial fault simulation.

2. ARCHITECTURAL EXPERIMENTS

We have three architectural experiments, Cosmic Cube, Mosaic, and Super Mesh, in various phases of design, construction, programming, and use. These machines are all ensembles of identical, concurrently operating, and regularly interconnected elements that communicate by message passing [5102:TR:83]. Our priority in these efforts has been to apply VLSI technology to achieve substantial advances in cost/performance in a limited set of computationally demanding tasks.

These experiments, the machine organizations, software systems, current status, and application span, can be summarized as follows:

2.1 Cosmic Cube

(W C Athas, Reese Fawcette, Mike Newton, Chuck Seitz)

Cosmic-cube is an experimental homogeneous machine with elements interconnected in a Boolean n-cube. Cosmic elements are of medium size for this class of machine, about 140 MSL, and consist of an Intel 8086 processor with 8087 floating point coprocessor, 128K bytes of primary storage, 8K bytes of read-only storage for initialization, bootstrap, and diagnostic programs, and 6 bidirectional self-timed communication channels.

At 140 million square lambda (MSL) complexity, and 78 "off the shelf" chips, the nodes of this machine are considered to be a hardware simulation of a node element that could be made as a single chip with 1 micron MOS technology. In anticipation of this advanced process technology, we have built this system in order to experiment with the applications, algorithms, and programming of such systems.

2.1.1 Current Status of the Hardware

Construction of the 6-cube (64 element) machine is now complete, and the system is in regular use. This machine has been completed and tested in stages of 3-, 4-, 5-, and 6-cubes in June, July, August, and September 1983, respectively, as node elements have been checked out.

A 2-cube (4 element) prototype has been running concurrent programs since July 1982, and has been used for software development. We are also operating an independent 3-cube machine for system software development.

The Cosmic 6-cube elements are running at a clock rate of 4.1 MHz, reduced from the interim design point of 5.0 MHz, due to speed problems in the Intel 8087 floating point coprocessor. As soon as all the 8087's are replaced by the -3 version, the system should operate at 5 MHz. Except for the 8087's, the system operates at up to 8 MHz. Accordingly, our current benchmarks can be expected to improve by a factor as much as 2 over the next year when faster 8087's become available, and due to an improved code optimizer.

Under separate support (principally DoE), production of about 200 nodes of a descendent of the cosmic cube design is under way at Caltech JPL, in

order to provide additional cycles for scientific users in the Caltech concurrent computation project. These nodes are software compatible with the cosmic 6-cube, and will assembled into a Boolean 7-cube (128 element) system and several smaller systems.

2.1.2 Application Programs and Benchmarks

An SU3 lattice gauge theory computation, an adaptation of a computation that had been run for about 1000 hours on the original 2-cube, is being used to test the 6-cube. This program, an investigation of the properties of protons predicted by the quantum chromodynamics theory, has now run for about 100 hours, and is producing successively more and more refined statistics. It will run for several hundred more hours before improving significantly on the best existing results obtained in about 40 hours on a Cray-1.

A LaPlace equation demonstration program that illustrates the relaxation solution in the progress of the computation is has been refined into a highly efficient and general program for differential equation solution by relaxation methods.

Both of these physics programs use substantially all of the node storage, and benchmark at 6:7 times the VAX11/780 on the present machine. At 8 MHz clock and by using a new code optimization package, we expect the Cosmic 6-cube to achieve more like 15 times the VAX11/780 for these regular computations, or easily in excess of 0.1 of a Cray-1.

There are numerous other application programs under development, the most interesting of which is a MOS-VLSI circuit simulator. The formulation that is used to achieve concurrency is a row partitioning of a modified nodal admittance matrix into concurrent processes [Mattisson 5096:DF:83]. A simulator working on this principle, written in Pascal, and running on a VAX, is the testbed for this program that will be transferred to the 6-cube this spring. The simulation formulation is described in more detail in section 4.2.

2.1.3 Software Status

The period of bringing up the 6-cube was one in which a large suite of testing and diagnostic programs have been written and refined. These programs are largely routine. The lowest level tests, such as the RAM test, are coded in 8086 assembly code, while the communication and floating point tests are coded in C.

The mature software tools for application programming of the Cosmic Cube now include a full initialization, bootstrap, and disagnostic package, a C and Unix based environment that is widely used for the more "crystalline" applications, and a complete Unix based simulator for programs written in this environment.

A prototype message passing and routing multiple process operating system called the "cosmic kernel" [5095:DF:83], has been defined and is being

coded and debugged. Since this will be the environment seen by people that might want to do software experiments with the Cosmic Cube after it is made available as a network server, an outline of the principal functions provided by and computational model imposed by the cosmic kernel (CK) is included in the concurrent computation section 3.2 below.

2.2 Mosaic Systems

(Chris Lutz, Steve Rabin, Chuck Seitz)

Mosaic is another experimental homogeneous machine, but with very small node elements, by the plan of this experiment, a single chip. This element consists of a mosaic processor with 4 input and 4 output ports (2.5 MSL) and as much primary storage as a single chip permits. For example, a 6 mm square chip in 3 micron MOSIS nMOS technology, 4000 lambda by 4000 lambda, 16 million square lambda (MSL), will accomodate a Mosaic processor, 4K bytes of RAM (32 bytes of which are "maimed" to provide a small initialization and bootstrap loader), and the small number of pads required for this element.

Mosaic elements can be interconnected in a variety of communication plans, including a tree, mesh, shuffle-exchange graph, chordal ring, or cube connected cycle.

A paper on the design of the Mosaic element, to be published in the Proceedings of the MIT Conference on Advanced Research in VLSI, January 1984, is included as Appendix A of this report, and is also available as a Caltech technical report [5093:TR:83].

MOSIS has fabricated a run of 48 prototype Mosaic processors for us, on which we got a 56% yield (27 working processors). These processors are being packaged on PCBs (designed with Earl and fabricated through the MOSIS prototype PCB service) with fast (InMOS) off-chip storage in order to make a working, programmable, and expandable 15 element Mosaic-tree or 16-element Mosaic-shuffle. These machines will be used for software development while we go through the logistics of building larger and more highly integrated versions of Mosaic systems. This staging tactic worked very well for the cosmic cube project.

An improved version of the processor with interrupts, a multiply instruction, a faster control PLA, and new microcode has been designed, but not yet verified. RAM test chips have been fabricated and tested. A full-size RAM element has been designed and verified, and is about ready to be sent to MOSIS for fabrication and test. Thus we believe we are very close to assembling a complete Mosaic element, a single chip with about 140,000 transistors.

This project involves a number of supporting efforts in testing to allow production of these chips in quantities of several thousands. We are putting a wafer-stepping probe station into operation in preparation for testing Mosaic elements on the wafers.

It is our intention in the runs of Mosaic element chips, both in an early run of about 20 wafers, and in a run later of sufficient wafers to yield about 1500 working chips, to work with Martin Buehler at JPL in correllating test strip results die by die with functional test results.

It is our plan to have a 1024-element Mosaic system running in June 1985.

A 1024-element Mosaic system is expected to be capable 2,500 Million instructions per second on combinatorial problems, or of 20:80 Million 32-bit mantissa floating point operations per second -- essentially Cray-1 performance -- on a limited class of matrix, grid point, and finite element computations. It is expected to exhibit a factor of about 10 in cost/performance over some of the most regular computations that can be performed on cosmic cube, and that do not require large amounts of storage per node, and a factor of about 100 in cost/performance over conventional mainframes for this limited set of problems.

Discussions of algorithms and programming systems for Mosaic are deferred to the concurrent computation section below.

2.3 Super-mesh

(Wen King Su, Chuck Seitz)

Super-mesh is a serial communication, serial floating point arithmetic, SIMD machine in the early stages of design. Its rationale was discussed in some length in our previous semi-annual report [5078:TR:83]. This machine might be regarded as a shared control implementation of a computational or systolic array.

The arithmetic algorithms and logic design for the super-mesh node are now complete, and the node fully simulated with MOSSIM.

The most substantial change made in the course of the design from the plans previously reported is a decision to use a 64-bit floating point format with a 56-bit mantissa and 8-bit exponent.

Based on early and partial layouts of the arithmetic slice and registers, the elements meet our previous size expectations, as scaled by the change in word size, to be about 2 MSL. Each element contains 40 registers, serial floating point arithmetic, neighbor communication, and the serial microcode receiver and pipeline. An instruction cycle of this machine requires 65 clock cycles. Since the serial carry-save arithmetic algorithms use only short combinational paths, we expect to clock this chip at 20 MHz, and achieve a floating point rate of 0.3 Mflops per element or 1.2 Mflops per chip with 4 elements per chip.

A microcode control word is transmitted serially for each instruction cycle. The physical design of this machine employs a deliberate skew in the internode communication and instruction broadcast to allow it to be extended to any size, but its interconnection is limited to a mesh.

2.4 Designs for Advanced Technology Homogeneous Machines

(Chuck Seitz)

A number of designs for an advanced technology homogeneous machine node element are being developed, with the following characteristics:

(1) 1 Mbyte of storage per node, with error correction, implemented with (40) 256K dRAM chips.

(2) 2 32-bit processors, one for communication and operating system functions, and the second for task processing (including fast floating point arithmetic), share the Mbyte of storage. Either the M68010 or DEC microVAX are possibilities for the processors.

(3) The communication section will be based in an evolution of the Fifo Buffered Tranceiver (FBT) chip previously reported [Ng 5055:TR:82]. The node will support up to 12 serial channels, which would allow up to 4096 element Boolean n-cube machines, and an additional channel for host, I/O, or secondary storage connections.

(4) The task section will use a floating point coprocessor with a floating point rate of 1 Mflop with "short" floating point words.

3. CONCURRENT COMPUTATION

3.1 Concurrent Algorithms

(Lennart Johnsson)

In the search for efficient algorithms for ensemble architectures, a few algorithms for sorting on binary trees and Boolean n-cubes, and for solving tridiagonal linear systems of equations on n-cubes have been devised [5085:DF:83]. The algorithms are totally distributed (as are the data structures).

Bitonic sort can be performed on a perfect shuffle network in (logN)*(logN) time, if there are one element per node. With one element per node n a Boolean n-cube the order of the time complexity is the same, but the constant can be improved somewhat. With several elements per node a combination of sequential sort and parallel sort is obviously necessary. A few algorithms have been devised for different sorting orders, and with different combinations of sequential/parallel sort. Some of the algorithms exhibit a gradual change of behavior from efficient sequential sort when only one processor is available to a Bitonic sort when there is as many processors as elements to be sorted. All nodes execute the same program. The control is entirely local.

The tridiagonal system solver devised for the n-cube solves the system in logN time if the cube is large enough that one dimension of the system to be solved can be identified with one node of the cube. If the system is larger than that, then the execution time grows lineraly, as on a sequential machine. The control is again entirely local, but somewhat more complex than in the sorting algorithms. The local control sequence can be derived from generators of a Gray code. What is effectively needed is to make successive linear embeddings in the cube, where the nodes in each embedding consists of nearest neighbors in the cube, and the succession of paths to be embedded are obtained by deleting every other node in the previous path.

3.2 Cosmic Kernel

(W C Athas, Reese Fawcette, Chuck Seitz)

The following is an outline of the principal functions provided by and computational model imposed by the cosmic kernel (CK), a small operating system kernel being developed for the cosmic cube.

One copy of the cosmic kernel (CK) resides in each node of the cosmic cube (CC), and all of these copies are concurrently executable. Some operating system functions are supported also in the CC intermediate host (IH). When running with this operating system, the IH does not run user code. It is dedicated to operating system and network functions, and the CC operates as a network server.

The kernel has two layers. All those parts of the kernel with which one communicates by system calls are in the "inner kernel" (IK). The inner kernel contains all message sending and receiving, message routing, and process scheduling.

All other kernel functions are invoked not by a system call mechanism, but by sending messages to a set of processes called the "outer kernel" (OK). The outer kernel provides capabilities of host I/O and of process creation and destruction.

The basic unit of the computations supported by CK is a "process." A single node of CC may contain many processes. A computation consists of a collection of processes distributed through the CC that can be thought of as all executing concurrently, either by virtue of being in different physical nodes of CC, or by being interleaved in execution within a single node. Processes communicate by sending and receiving messages.

The placement of processes in physical nodes of CC can be controlled by the programmer, or may be deferred to a library process. This placement does not influence the logic of the program, but will have consequences in (1) the possibility of exceeding available storage, (2) the influence of process placement on performance through the overhead in message routing and the competition for cycles amongst processes in a single node.

As far as the kernel is concerned, a process is a segment of sequential code and data of fixed size. This code and data is represented for communication purposes as a binary image relocatable by the segmentation features of the 8086 processor. Process code must be dynamically relocatable; it must not load or manipulate the code segment (CS), data segment (DS), or stack segment (SS) registers, and must maintain a stack with sufficient space for storing state in an interrupt. The code, data, and stack segments are each limited to 64K bytes.

The code for a process is written in a suitable programming notation, such as extended versions of Pascal, C, or 8086 Assembly, and compiled independently of other processes that may be a part of the same computation. Because of the independence of the construction of process code, and the standardization of kernel functions, there is complete and uniform compatiblity of processes independent of source language. For example, a library process written in C or assembly code can be used in a computation in which most of the processes were written in Pascal.

Each process has a unique 16-bit identifier that is an ordered pair: process id = <physical node, process number within the node>. This id is normally represented in a single 16 bit word, in which the physical node has a range 0:255 and the process number a range 0:255.

Because the physical location of a process is imbedded in its id, CK does not maintain a map from process id to physical node. Message routing to processes is based simply on the physical address part of the destination found in the message header. Thus we assume that a process, once created, is not relocated.

CK supports one message format. All messages have headers. Long messages are communicated over the physical channels of CC by different protocols than are used for short messages, but this difference is invisible to user programs.

The message header can be thought of as the envelope in which a message is sent. The header is 4 words long, and contains

word 1: id of the destination process
2: id of the sending process
3: message type
4: message length

Words 1 and 2 are self-explanatory. The 16-bit type is significant in the way in which messages sent are matched against messages expected by the destination process. The message length is specified in number of words, and may be zero. A message of length 0 is called a "synchronization message," and conveys information only through the type and by its existence.

A process performs message sending and receiving by system calls. The system call is implemented on the cosmic node as a software interrupt. For present purposes calls will be described by a name followed by parameters, if any. In C or Pascal source, system calls may be either calls to an external procedure that includes the system call, or may be compiled directly into the suitable system call.

The two basic message calls are SEND and RECV. These calls specify a communication request that will be satisfied as soon as CK is able. CK buffers messages in transit, up to and including complete messages. It is perfectly legal to SEND a message before a corresponding RECV is executed; such a message will be queued in the destination node and in transit as storage space allows. CK also supports a PROBE call that checks for the presence of a message queued for a process, and an UNRECV call that will undo a pending RECV. Details of these functions are described in [5095:DF:83].

The outer kernel (OK) is a priviledged set of processes whose functions are invoked by messages rather than system calls. User processes will normally communicate with their own OK processes, but may equally communicate with any other OK processes.

If we might be allowed to indulge briefly here in design philosophy, let us remark that any future evolution of CK is seen as occurring in the outer kernel. The number of functions of the inner kernel, accordingly its size, complexity, and difficulty in portability, has been guarded fairly closely. These functions are very close to machine intrinsics, and may well guide the development of future node architectures. The outer kernel is meant to be more nearly open-ended, machine independent, and accessible to change.

The set of process ids <*,255>, <*,254>, ... is reserved for the OK. Processes <*,254> have capabilities of performing I/O with hosts, and so are denoted <*,host>. Processes <*,255> have capabilities of creating and destroying processes, and the associated capability of performing storage management within a own node, and are denoted <*,spawn>.

3.3 Mosaic Software

3.3.1 Scheduler

(Pey-yun Peggy Li, Lennart Johnsson)

A Scheduler has been implemented in Mosaic Assembly Language and tested under the Simulator. The scheduler keeps track of the states of the processes, i.e., RUN, READY, SUSPEND and SLEEP states. The state transitions are triggered either by the running process while an I/O operation fails or by the scheduler while it receives a message at one input port. The scheduler occupies 387 words of memory and one Process Control Block takes 30 words. The time to perform a context switch (swap in and swap out) and inspect all the three input ports is about 150 instruction cycles.

3.3.2 Tree Downloader

(Pey-yun Peggy Li, Lennart Johnsson)

A Multi-node Downloader for a Mosaic-tree with the mapping algorithm [5084:TR:83] implemented in it has been written and tested. The downloader can load a fixed number of process programs into one Mosaic element. That number is furnished by the host and propagated through the entire tree. The downloader has three parts, initiation, type loading and name loading. The initiation part creates, allocates and initiates the process control blocks for the fixed number of processes.

The type loading part loads the proper number of node types into each node, and the code loading part loads the relocatable program code of all the residing processes into each node's memory. The type loading part is running in time sharing mode for simplicity reason. Because of the heavy context switches, the root processor takes about 13,000 instruction cycles to load the node type string of a five level tree into a four level tree machine. Meanwhile, it takes about 10,000 instruction cycles to load all the program code down into the tree, provided that the root node contains two different node types and there are totally five different node types, one for each level, and each program is 200 words long.

For an L level binary tree which is mapped onto a M level tree machine, L > M, the time to load the node type string at the root processor can be formulated as follows:

$$T = 7000 + \{2 * \sum_{i=4}^{n+1} [2^{**}(i-1) - 2^{**}(i-3) - 1] * 1000\} + N * 1000 * (L-n-1)$$

where n = M-L and N = 2**n-1, eg, the number of nodes shared in the root processor.

The documentation for the scheduler and the multi-node downloader is in preparation.

3.3.3 A Modic Compiler for Mosaic

(Alain Martin)

We are building a compiler for translating a high-level language for distributed computations into Mosaic code.

According to the principle that "a complex system that works is invariably found to have evolved from a simple system that worked" (John Gall), we have decided to start with a simple language called Modic. The sequential part of the language is based on Dijkstra's guarded commands. Communication and synchronization are provided by input and output commands (similar to CSP's) on channels. A channel is a programming concept that makes it possible to match an input command in one process to an output command in another process. A Boolean operation on a channel, called the "probe", allows one to test whether an input or output command is pending on the channel.

Later, the language will be extended with procedures, dynamic creation of processes, multiple channels, (i.e., channels shared by more than two processes), select, and broadcast operations.

4. VLSI DESIGN

4.1 Switch Simulation Tools

4.1.1 FMOSSIM

(Mike Schuster, Randy Bryant)

FMOSSIM, a fault simulator for MOS digital systems first became operational in April, 1983. This program utilizes the same switch-level representation of MOS circuits as the logic simulator MOSSIM II. As a consequence, it can model such MOS circuit structures as (bidirectional) pass transistors, static and precharged logic, busses, and both static and dynamic memory. Faults are represented as alterations of the switch-level description causing selected nodes to be stuck-at 0 or 1, or selected transistors to be stuck open or closed. Faults such as breaks in wires or short circuits between wires can also be modeled by adding extra "fault" transistors to the network description.

This combination of circuit and fault modeling capabilities is far more general than has been achieved previously. FMOSSIM utilizes concurrent simulation techniques to simultaneously model the good circuit and a large number of faulty circuits, and consequently requires much less CPU time than simple serial fault simulation. Both the utility and the performance of this program seem quite promising.

A paper on FMOSSIM, to be published in the Proceedings of the MIT Conference on Advanced Research in VLSI, January 1984, is included as Appendix B of this report, and is also available as a Caltech technical report [5101:TR:83].

4.1.2 The MOSSIM Simulation Engine

(Bill Dally, Randy Bryant)

We have begun the design of the Mossim Simulation Engine (MSE), a special purpose processor for performing switch level simulation of MOS VLSI circuits [5100:TR:83]. A single MSE processor will be constructed from ~400 TTL MSI and MOS memory devices packaged on a single 15" x 15" wire-wrap board, and will perform switch level simulation at a rate of ~5*10^5 logic events per second, 500 times faster than MOSSIM II running on a VAX-11/780. Several MSE processors may be connected in parallel to achieve additional speedup.

4.2 Circuit Simulation on the Cosmic Cube

(Sven Mattison, Lennart Johnsson, Chuck Seitz)

We have completed a study of MOS-VLSI circuit simulation formulations for concurrent execution [5096:DF:83].

This effort is motivated by two threads of our research. First, SPICE uses lots of time on our computers as well as on many people's supercomputers, and such a program would provide a more economical way to doing these simulations. The second reason is that circuit simulation is an excellent paradigm of a computation with concurrency opportunities in which the communication graph, while fixed, is not so regular as in the computations being done by our collaborators in the sciences at Caltech.

The formulation we have chosen, a modified nodal admittance matrix, is not as general as the usual circuit simulator, and will not treat elements such as ideal opamps, current controlled current sources, ideal transformers, or nonlinear elements that lack an unambiguous admittance description. This formulation is, however, perfectly adequate for MOS-VLSI, and offers some storage and performance advantages over more general formulations.

The most critical aspect of the approach to concurrent execution is the sparse matrix equation solution method. Direct solution methods are hard to implement on a machine such as the cosmic cube, and iterative methods very natural. The most common iterative methods are the Jacobi (J), Gauss-Seidel (GS), successive overrelaxation (SOR), and the conjugate gradient (CG) method.

Among these basic iterative methods, J, GS, and SOR use only one row at a time in the matrix to calculate a new estimate for each component in the unknown vector. Thus there is both locality and concurrency achieved by row partitioning. SOR requires an accurate estimate of the relaxation parameter to be efficient, and its local calculation is sufficiently difficult to eliminate SOR from consideration. CG is not strictly iterative, and because it uses matrix inner products during each iteration is less local than J or GS.

When choosing a method for solving the matrix equation, one must weigh also that the matrix solving on sequential machines represents only 10:20% of the cycles, while the more freely concurrent model equation evaluation represents 80:90% of the cycles. One prefers that the partitioned matrix equation solving algorithm have an interface to the model evaluating routines that causes as little communication and redundant model evaluation as possible. There is a good correspondence between the elements and the row partitioning. Each row in the modified nodal admittance representation contains only entries from devices connected to the node represented by the row.

In order to see if iterative methods are considerably slower than direct ones, SPICE 2 version G.5 was modified to use GS iterations in the transient analysis. GS was used instead of Jacobi, since it does not need extra storage for the new estimates of the unknowns, and is easier to code, but since GS and J have similar convergence properties, the results should apply to both methods. Three different test circuits were used, with the consistent result that the performance was nearly identical.

Thus, based on these studies, the circuit simulator we are writing for the

cosmic cube will use a row partitioned matrix equation solving routine. Among the iterative solution methods, the Jacobi iteration assures convergence independent of the order in which the different processes complete their calculations)for matrices conditioned in a way that is easily assured for MOS circuits). Within each process the modified Newton-Raphson method is used as the inner iteration of the global Jocobi iteration. A first or second order predictor-corrector backward differentiation formula is assumed as the integration algorithm.

4.3 From Circuit to Layout - Another Approach

(Tak-Kwong Ng, Lennart Johnsson)

The circuit embedding problem can be transformed into the problem of graph embedding. A proper graph model for studying MOS circuit layout topology has been proposed, and an alogorithm for mapping a circuit into its graph model has been implemented.

Transistors whose gates belong to the same net are lumped together to form one component. This component is represented by a vertex. Each individual source or drain is mapped into an edge incident on this vertex. All other connections to the gate net are represented by edges incident on this vertex.

Serial transistor configurations can be considered as a transistor with multiple gates. Such component is represented also by a vertex. The source and drain are mapped into edges incident on this vertex. All gate connections are represented by edges incident on this vertex.

The exterior of a layout is also mapped into a vertex. For each external net, there is an edge connecting the exterior vertex and the external net vertex.

Hopcroft and Tarjan's graph planarity testing algorithm is modified to include the procedure for saving the appropriate information which is necessary for deriving the proper embedding. The information is saved as constraint graphs. Thus, the graph embedding problem becomes the problem of assigning edges to some plane such that all constraints are met, and a predefined function is optimized. Optimal solutions can be found by enumeration. Several heuristic approaches are being studied.

4.4 Mosaic RAM element design

(Steve Rabin, Chris Lutz, Chuck Seitz)

A RAM section for the Mosaic element has been designed. Each RAM section stores 4K bits, and 8 copies of this macrocell will be used in the 16 MSL version of the Mosaic element.

Although specialized processes provide higher storage density than

processes that are suitable for the Mosaic processor, a processor with on-chip memory on has many advantages over processor and memory in separate packages. These advantages include many times lower volume, pin count, signal energy, and driver delay, resulting primarily from the integration of the memory bus into a single package.

The Mosaic memory was designed to be quite process technology independent, and able to benefit from nMOS technology to under 2um feature size. It was also designed to take full advantage of and extend the circuit style of "hot clock" bootstrap drivers used in the Mosaic processor. The final design uses exactly no depletion transistors.

The memory must be very fast, fast enough to perform an access at least every 300 tau, and have a 16-bit word interface. The memory must have a negligible soft error rate, and would ideally dissipate negligible DC power. The memory must be flexible enough to be configured in various sizes of up to the 4K words (64K bits) of processor address space.

Process independence and organizational simplicity lead us to select a two-bus three-transistor dynamic RAM memory cell design.

Commercial single transistor dynamic memories require dynamic node refresh every 2ms. Systems using such devices typically use error detecting/correcting codes to bring soft errors to acceptable levels. It is expected that the use of many banks of concurrently refreshing three transistor cells with fairly large storage nodes combined with the 50 usec refresh period provided by Mosaic processor will provide very good immunity to soft errors.

The memory bus is pipelined to take advantage of the rather generous memory latency permitted by the processor. Allowable instruction latency is one cycle plus one phase. Allowable data or write latency is two complete cycles. Pipelining in this fashion allows us to reduce the bandwidth to/from the processor to 29 bits/cycle (1 write, 12 address, 16 data).

Each memory access starts with a column read followed almost always by a write to the same column on the next cycle. Because this write occurs in the same cycle as the next read, the storage control is somewhat tricky. Mandating a write causes one of several words from the memory to be replaced with write data from the processor. If another word on the same column is accessed on that cycle, the subsequent write back to the column would permanently store incorrect (stale) data. For this reason a write back override is enabled on the second cycle after a write cycle.

Two subsidiary clock phases, ph1lL and ph12L have been introduced, and divide the processor cycle into 6 epochs. A conservative clocking discipline is used to permit switch level simulation using a unit delay timing model.

Circuit design for the Mosaic storage element avoids depletion loads due to associated scaling and process problems incompatible with the goals of

aggressive technology independence and low DC power consumption. Dynamic logic is used extensively. Race conditions are avoided entirely and charge sharing is kept to a minimum. Large capacitive loads are admirably driven by rising edge logic using bootstrap clock drivers combined with precharged logic devices typically underneath the memory bus itself. Circuit simulations predict operation at 15 MHz to match the predicted performance of 3 micron Mosaic processors.

So, our objectives have been met by a conceptually simple, scalable memory, optimized for advancing MOS process technology. This memory design imposes the following domain restrictions:

Consecutive write operations are not supported.

Read operations immediately following a write operation will not refresh the dynamic storage nodes of the column so read.

Reading a word that was written the previous cycle is not supported.

The first two conditions do not occur in the microcode, and the third condition occurs only by writing into the instruction stream.

4.5 SOS technology, PCB technology

(Chuck Seitz)

A fairly extensive cell library for CMOS/SOS, all in Manhattan geometry in order to be compatible with design tools that do not handle real geometry, has been provided to MOSIS for distribution. Circuits received from the first MOSIS SOS run have been tested, and the process behaves exactly as expected. A new SOS technology writeup is in preparation for MOSIS distribution.

We were happy to have assisted Ms Mosis in the development of PCB services. Earl and our various plotting programs have been modified to accommodate PCB technology, and we have made use of this service as indicated in section 2.2 to package a prototype Mosaic machine. The PCBs received very quickly from MOSIS were unremarkable except for a MOSIS logo of excessive size.

California Institute of Technology
Computer Science, 256-80
Pasadena, California 91125

ARPA Technical Memos and Technical Reports
together with selected other Caltech reports on VLSI topics
November 1983

Available from the Computer Science Department Library

+++

5102:TR:83 "Experiments with VLSI Ensemble Machines," October 1983;
Seitz, Charles L.

5101:TM:83 "Concurrent Fault Simulation of MOS Digital Circuits,"
October 1983
Bryant, Randal E.

5100:DF:83 "MOSSIM Simulation Engine Preliminary Architecture," October
1983;
Dally, Bill and Randy Bryant

5099:TM:83 "VLSI and the Foundations of Computation," September 1983;
Mead, Carver

5098:TM:83 "New Techniques for Ray Tracing Procedurally Defined
Objects," September 1983;
Kajiya, James T. [$2.00] 13 pages

5097:TR:83 "The Design of a Self-timed Circuit for Distributed Mutual
Exclusion," September 1983;
Martin, Alain J.

5096:DF:83 "MOS-VLSI Circuit Simulation Formulations for Concurrent
Execution," August 1983;
Mattisson, Sven

5095:DF:83 "A System Programmer's Guide to Cosmic Kernel ," August 1983;
Seitz, Chuck and Athas, Bill

5093:TR:83 "Design of the MOSAIC Element," July 1983;
Lutz, Chris, Steve Rabin, Chuck Seitz and Don Speck 10
pages

5092:TM:83 "Residue Arithmetic & VLSI," July 1983;
Chiang, Chao-Lin & Lennart Johnsson [$2.00] 5 pages

5091:TR:83 "Race Detection in MOS Circuits by Ternary Simulation," June 1983;
Bryant, Randal E. [$2.00] 12 pages

5090:TR:83 "Space-Time Algorithms: Semantics and Methodology,"Ph.D. Thesis, June 1983;
Chen, Marina Chien-mei [5.00] 109 pages

5089:TR:83 "Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circuits," July 1983;
Lin, Tzu-Mu and Carver A. Mead [$10.00] 68 pages

5088:TM:83 "infinite Fair Shuffle," June 1983;
Choo, Young-il

5087:TM:83 "Concurrency Algebra and Petri Nets," June 1983;
Choo, Young-il

5086:TR:83 "A VLSI Combinator Reduction Engine,"MS Thesis May 1983;
Athas, William C., Jr. [$4.00] 40 pages

5085:DF:83 "Solution Set of AM/CS146," May 1983;
Johnsson, Lennart and Peggy Li

5084:TM:83 "The Tree Machine: An Evaluation of Strategies for Reducing Program Loading Time," August 1983;
Li, Pey-yun Peggy, and Lennart Johnsson [$2.00] 26 pages

5083:DF:83 "Role of Parameters-Sticks Representations," May 1983;
Trimberger, Steve

5081:TR:83 "RTsim - A Register Transfer Simulator," April 1983;
Lam, Jimmy [$4.00] 63 pages

5079:TR:83 "Highly Concurrent Algorithms for Solving Linear Systems of Equations," April 1983;
Johnsson, Lennart [$2.00] 79 pages

5078:TR:83 "Submicron Systems Architecture," April 1983;
ARPA Semiannual Technical Report [$5.00] 32 pages

5077:DF:83 "Pooh: A Uniform Representation for Circuits," March 1983;
Whitney, Telle

5076:DF:83 "The Semantics of a Functional Language for VLSI Systems," March 1983;
Chen, Marina

5075:TR:83 "A General Proof Rule for Procedures in Predicate Transformer Semantics," March 1983;
Martin, Alain [$2.00] 20 pages

5074:TR:83 "Robust Sentence Analysis and Habitability," Trawick, David J., Ph.D. Thesis, February 1983

5073:TR:83 "Automated Performance Optimization of Custom Integrated Circuits," Trimberger, Stephen, Ph.D. Thesis, March 1983

5068:TM:83 "A Hierarchical Simulator Based on Formal Semantics," Proc. Third Caltech Conf. on VLSI, p. 207-223, March 1983; Chen, Marina and Carver Mead [$1.00]

5065:TR:82 "Switch Level Model & Simulator for MOS Digital Systems," December 1982; Bryant, Randal E. [$3.00]

5059:TM:82 "A Comparison of MOS PLAs," December 1982; Trimberger, Stephen [$2.00]

5055:TR:82 "FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems," MS Thesis, December 1982; Ng, Charles H. [$5.00]

5052:TR:82 "Submicron Systems Architecture," submitted to ARPA October 1982; Semiannual Technical Report [$4.00]

5049:TR:82 "Distributed Mutual Exclusion Algorithms," submitted for publication, AJM 31, September, 1982; Martin, Alain [$3.00]

5047:TR:82 "The Torus: An Exercise in Constructing a Processing Surface," Proc. 2nd Caltech Conference on VLSI, Caltech, Pasadena CA, January 1981; Martin, Alain [$3.00]

5046:TR:82 "An Axiomatic Definition of Synchronization Primitives," Acta Informatica 16, pp. 219-235 (1981); Martin, Alain [$3.00]

5045:TM:82 "A Distributed Implementation Method for Parallel Programming," Proc. Information Processing '80, S. H. Lavington, (ed.); Martin, Alain [$3.00]

5044:TR:82 "Hierarchical Nets: A Structured Petri Net Approach to Concurrency," September, 1982; Choo, Young-Il [$10.00]

5043:TM:82 "A Formal Derivation of Array Implementations of FFT Algorithms," Proc. USC Workshop on VLSI & Modern Signal Processing, (sponsored by ONR) Nov. 1982, to be published by Prentice-Hall; Johnsson, Lennart and Danny Cohen [$3.00]

5042:TR:82 "Concurrent Algorithms as Space Time Recursion Equations," September, 1982; Chen, Marina and Carver Mead [$4.00]

5040:TR:82 "Concurrent Algorithms for the Conjugate Gradient Method," September, 1982; Johnsson, Lennart [$4.00]

5038:TM:82 "A New Channel Routing Algorithm," September, 1982; Chan, Wan S. [$4.00]

5035:TR:82 "Type Inference in a Declarationless, Object-Oriented Language," MS Thesis, June 1982; Holstege, Eric [$10.00]

5034:TR:82 "Hybrid Processing," Ph.D. Thesis, March 1982; Carroll, Chris [$12.00]

5033:TR:82 "MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual," August 1982; Schuster, Mike and Randal E. Bryant [$4.00]

5030:TM:82 "VLSI Algorithms for Doolittle's, Crout's, and Cholesky's Methods," Proc. ICCC '82, IEEE Int'l Conf. on Circuits & Computers, NY Sept. 1982, pp.372-377, IEEE Catalog No. 82CH1813-5; Johnsson, Lennart [$1.00]

5029:TM:82 "POOH User's Manual," July 1982,; Whitney, Telle [$4.00]

5027:TM:82 "Concurrent Programming," July 1982; Bryant, Randal E. and Jack B. Dennis [$3.00]

5021:TR:82 "Earl: An Integrated Circuit Design Language," MS Thesis, May 1982; Kingsley, Chris [$5.00]

5019:TR:82 "A Computational Array for the QR-Method," Proc. MIT Conference on Advanced Research in VLSI, P. Pennfield, ed., Boston, January 1982, pp.123-129; Johnsson, Lennart [$3.00]

5018:TM:82 "Filtering High Quality Text for Display on Raster Scan Devices," August 1981; Kajiya, Jim and Mike Ullner [$2.00]

5017:TM:82 "Ray Tracing Parametric Patches," May 1982; Kajiya, Jim [$2.00]

5016:TR:82 "Bristle Blocks - Scrutinized and Analyzed," June 1982; McNair, Richard, and Monroe Miller [$4.00]

5015:TR:82 "VLSI Computational Structures Applied to Fingerprint Image Analysis,"; Megdal, Barry [$4.50]

5014:TR:82 "The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture," Ph.D Thesis, April 1982; Lang, Charles R. Jr., [$15.00]

5012:TM:82 "Switch-Level Modeling of MOS Digital Circuits,"; Bryant, Randal [$2.00]

5005:TM:82 "Chip Assembly Tools,"; Trimberger, Steve and Chris Kinglsey [$2.00]

5004:TM:82 "Riot - A Simple Graphical Chip Assembly Tool,"; Trimberger, S. and Jim Rowson [$2.00]

5003:TM:82 "Pipelined Linear Equation Solvers & VLSI ," Proc., Microelectronics 1982, Adelaide, Australia, May 1982, pp.42-47, The Institution of Engineers, Australia, Nat'l Conf. Pub. No. 82/4; Johnsson, Lennart S. [$2.00]

5001:TR:82 "Minimum Propagation Delays in VLSI ," IEEE J. Solid State Circuits, Vol. SC-17, No. 4, August 1982, pp. 773-775; (later version of 4601:TM:81); Mead, Carver, and Martin Rem [$2.00]

5000:TR:82 "A Self-Timed Chip Set for Multiprocessor Communication," MS Thesis, February 1982; Whiting, Douglas [$6.00]

4778:TM:82 "Testing and Structured Design," Proc. International Test Conference., Cherry Hill NJ, August 1982; DeBenedictis, Erik P., and Charles L. Seitz [$2.00]

4777:TR:82 "Techniques for Testing Integrated Circuits,"; DeBenedictis, Erik P.

4724:TR:82 "Concurrent, Asynchronous Garbage Collection Among Cooperating Processors," (superceded by 5014:TR:82); Lang, Charles R. [$5.00]

4716:TM:82 "A Rectangular Area Filling Display System Architecture,"; Whelan, Dan [$4.00]

4710:TM:82 "Earl: An Integrated Circuit Design Language," (Succeeded by 5021:TR:82); Kingsley, Christopher [$5.00]

4684:TR:82 "A Characterization of Deadlock Free Resource Contentions,"; Chen, Marina, Martin Rem, and Ronald Graham [$3.00]

4682:TR:81 "Earl: An Integrated Circuit Design Language," (Succeeded by 5021:TR:82); Kingsley, C. [$3.00]

4675:TR:81 "Switching Dynamics," MS Thesis,; Lewis, Robert K. [$7.00]

4655:TR:81 "Proceedings Second Caltech Conference on Very Large Scale Integration," 19-21 January 1981; Seitz, Charles, ed. [$20.00]

4654:TR:81 "A Versatile Ethernet Interface," MS Thesis; Whelan, Dan [$12.00]

4653:TR:81 "Toward A Theorem Proving Architecture," MS Thesis; Lien, Sheue-Ling [$10.00]

4618:TM:81 "The Tree Machine Operating System,"; Li, Peggy [$5.00]

4601:TM:81 "Minimum Propagation Delays in VLSI," Proc., Second Caltech Conf. on VLSI, January 1981; Mead, Carver A. and Martin Rem [$3.00]

4600:TM:81 "A Notation for Designing Restoring Logic Circuitry," Proc., Second Caltech Conf. on VLSI, January 1981; Rem, Martin, and Carver A. Mead, (revised from 4529:TM:81) [$3.00]

4530:TR:81 "Silicon Compilation," Ph.D. Thesis; Johannsen, Dave [$18.00]

4521:TR:81 "Lambda Logic," MS Thesis; Rudin, Leonid [$8.00]

4517:TR:81 "The Serial Log Machine," MS Thesis; Li, Peggy [$7.00]

4407:TM:82 "An Experimental Composition Tool,"; Mosteller, Richard C. [$3.00]

4379:TR:81 "LAP User's Manual,"; Lang, D. (Rev. by C. Kohle and S. Trimberger 9/81) [$2.00]

4336:TR:81 "A Structured Design Methodology and Associated Software Tools,"; Trimberger, S., J. Rowson, D. Lang, and J.P. Gray [$4.00]

4334:TR:81 "An Inexpensive Multibus Color Frame Buffer,"; Whelan, Dan and R. Eskanazi [$1.00]

4332:TR:81 "RLAP, Version 1.0, A Chip Assembly Tool,"; Mosteller, R. [$3.00]

4320:TR:81 "A Hierarchical Design Rule Checker," MS Thesis; Whitney, Telle [$7.00]

4317:TR:81 "REST - A Leaf Cell Design System," MS Thesis; Mosteller, Richard C. [$10.00]

4298:TR:81 "From Geometry to Logic," MS Thesis,; Lin, Tzu-mu [$5.00]

4287:TR:81 "Computational Arrays for Band Matrix Equations,"; Johnsson, L. [$2.00]

4281:TR:81 "Combining Graphics and a Layout Language in a Single Interactive System (2nd Revision),"; Trimberger, S. [$2.00]

4270:TR:81 "FIFI Test System Preliminary User's Manual," (later version contained in 4777:TR:82); DeBenedictis, Erik [$3.00]

4204:TR:78 "A 16-Bit LSI Digital Multiplier," EE Thesis; Masumoto, R. T. [$8.00]

4191:TR:81 "Towards A Formal Treatment of VLSI Arrays," Proc., Second Caltech Conference on VLSI, Pasadena, CA, January 1981; Johnsson, Lennart S., Uri Weiser, D. Cohen, and Alan L. Davis [$4.00]

4168:TR:81 "Computational Arrays for the Discrete Fourier Transform," Proc., 22nd Computer Science Int'l. Conference, CompCon 81, San Francisco, February 1981, pp. 236-244, IEEE Catalog No. 81CH1626-1; Johnsson, Lennart S. and D. Cohen [$3.00]

4088:TR:80 "The Representation of Communication and Concurrency,"; Milne, George [$8.00]

4087:TR:80 "Gaussian Elimination of Sparse Matrices and Concurrency - A Complexity Analysis," (Succeeds 4087:DF:80); Johnsson, Lennart [$3.00]

4061:TR:80 "A Preliminary Report on the Caltech ARPA Tester Project," (later version in 4777:TR:82); DeBenedictis, Erik [$4.00]

4029:TR:80 "Structure, Placement and Modeling," MS Thesis; Segal, R. [$8.00]

4025:TM:80 "Sticks Standard Software Package,"; Segal, R. and Steve Trimberger [$2.00]

4024:TM:80 "SSP Basic Software Package (Revised),"; [$5.00]

4022:TM:80 "Comprehensive CIF Test Set (Revised),"; Trimberger, S. [$5.00]

3901:TM:78 "Hierarchical Design for VLSI,"; Rowson, J. [$3.00]

3882:TM:80 "A Chip Assembler,"; Tarolli, Gary [$2.00]

3880:TM:80 "The Proposed Sticks Standard,"; Trimberger, S. [$5.00]

3857:TM:80 "VLSI Architecture & Design," Proc., National Electronics Conference, Chicago, Oct., 1980, Vol. 34, pp. 254-259; Johnsson, Lennart [$1.00]

3805:TR:80 "SSP Annual Report,"; Silicon Structures Project [$3.00]

3762:TR:80 "A Software Design System," Ph.D. Thesis; Hess, Gideon [$8.00]

3761:TR:80 "A Fault Tolerant Integrated Circuit Memory," Ph.D. Thesis; Barton, Tony [$7.00]

3760:TR:80 "The Tree Machine: A Highly Concurrent Computing Environment," Ph.D. Thesis,; Browning, Sally [$10.00]

3759:TR:80 "The Homogeneous Machine," Ph.D. Thesis; Locanthi, Bart [$7.00]

3710:TR:80 "Understanding Hierarchical Design," Ph.D. Thesis; Rowson, James [$9.00]

3642:TM:80 "Modeling and Verification in Structured Integrated Circuit Design," Ph.D. Thesis; Buchanan, Irene [$10.00]

3487:TM:80 "The Proposed Sticks Standard,"; Trimberger, Steve [$2.00]

3364:TM:79 "Stack Data Engine," December 1979; Efland, G. and R. C. Mosteller [$5.00]

3357:TM:79 "CIF2OP Instruction Manual,"; Tarolli, Gary and Dick Lang [$3.00]

3356:TR:79 "LAP User's Manual,"; Lang, Dick [$3.00]

3353:TM:79 "FORTRAN Debugging Aids,"; Trimberger, Steve [$3.00]

3352:TM:80 "A Comprehensive CIF Test Set," December 1979; Trimberger, Steve [$2.00]

2883:TR:79 "A Pascal Machine Architecture Implanted in Bristle Blocks, a Prototype Silicon Compiler," MS Thesis; Seiler, Larry [$10.00]

2870:TM:79 "A Wire Oriented Mask Geometry Editor,"; Trimberger, Steve [$5.00]

2686:TR:80 "The Caltech Intermediate Form for LSI Layout Description,"; Sproull, Robert and Richard Lyon, revised by S. Trimberger [$2.00]

2276:TM:78 "A Language Processor and a Sample Language,"; Ayres, Ron [$12.00]

1584:TM:78 "Cost and Performance of the VLSI,"; Mead, Carver A. [$3.00]

1438:TM:78 "Polygon Package,"; Sutherland, Ivan E. [$3.00]

DESIGN of the MOSAIC ELEMENT

Chris Lutz, Steve Rabin, Chuck Seitz, Don Speck

Department of Computer Science
California Institute of Technology
Pasadena, California 91125

ABSTRACT

The Mosaic element is a fast single chip computer designed to be used in groups for concurrent computation experiments. Each element contains a 16-bit processor, read-write storage, read-only store for a small initialization and bootstrap loading program, four input ports, and four output ports. The Mosaic processor, a highly structured design that achieves very good performance and density through innovations in its microcode, circuit techniques, and layout, is described in detail.

INTRODUCTION

Myriads of Mosaic elements can be connected together by their ports in a variety communication plans to form a family of specialized, high performance, concurrent, and programmable computing engines. In addition to its end use as a component for experiments with concurrent computing engines, the Mosaic element has been an interesting vehicle for numerous adventures in VLSI design, design tools, and testing. It includes experiments and innovations in its microcode, circuit techniques, and layout, with performance being a central objective throughout.

A Mosaic element with 4K bytes of read-write storage, approximately 140K transistors on a chip 4000 lambda square (6 mm square at 3 micron feature size), is sufficiently complex to have given our design tools a thorough workout, and have stretched our capabilities for laying out, verifying, and testing large structured designs.

The original models for this project were (1) Sally Browning's research on algorithms for a programmable tree machine[1,2,3] and (2) the "OM" described in Mead & Conway[4]. Mosaic started out as a tree machine element, but we have since come to see it as a building block for a variety of fine grain ensemble machines[5] with connection plans up to degree four, such as a tree, mesh, shuffle, chordal ring, or cube connected cycle. The influence of the OM2 on the processor datapath layout is apparent.

Several early attempts to lay out a much less ambitious processor with a 4-bit path to off-chip storage managed to break our design tools, and were thus indirectly the origin of the constraint solving composition and geometry tool Earl[6] used for the present design. A new processor with a 16-bit path to storage that could be placed on-chip was designed in 1982, sent to MOSIS in January 1983, and functioned essentially correctly, and at 7 MHz (4 micron feature size), on first silicon in February 1983. The processor design was subsequently augmented to include additional functions, to speed up the control PLA and to incorporate the planned on-chip storage. It is this latest design that is described here.

TOP LEVEL VIEW

It appears that most of the silicon area in multiple instruction multiple data (MIMD) ensemble machines will be devoted to storage. In Cosmic Cube[7], a larger grain size machine at Caltech whose organization is otherwise similar to Mosaic, the fraction of the element complexity devoted to storage is about 75%. With the precondition that a complete Mosaic element fit on a single chip, and using today's MOSIS nMOS fabrication with 1.5 micron lambda (3 micron feature size) on chips 6 mm square, the complexity of today's Mosaic element is limited to 4000 by 4000 lambda, or 16 million square lambda (MSL). This area is apportioned with about 2.5 MSL for the processor and ports, 1 MSL for the pad frame, and 12 MSL (75%) for storage and its interconnect. The area allowed for the processor is quite small, less than 6 sq mm, or 9,000 sq mils.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

The storage is partitioned into several smaller arrays, as suggested by the analysis presented in section 8.5 of Mead & Conway[4]. Each array is 4096 bits, 64 by 64, organized to interface with the processor as 256 16-bit words. The densest read-write storage we understand how to make with MOSIS nMOS technology is based on a 3-transistor dynamic storage cell, which requires that this storage be refreshed periodically. This refresh function is accomplished in the microcode of the processor. The very small amount of read-only storage required for the initialization and bootstrap loader is implemented with a set of "maimed" RAM cells.

Thus the 16 MSL Mosaic has the floorplan shown in figure 1, but if more MSL were made available by a reduction in lambda, one could use this area to pack in more storage. While the processor is only 16% of the area of the chip, it represents about 90% of the design effort, so most of the following description concentrates on the processor.

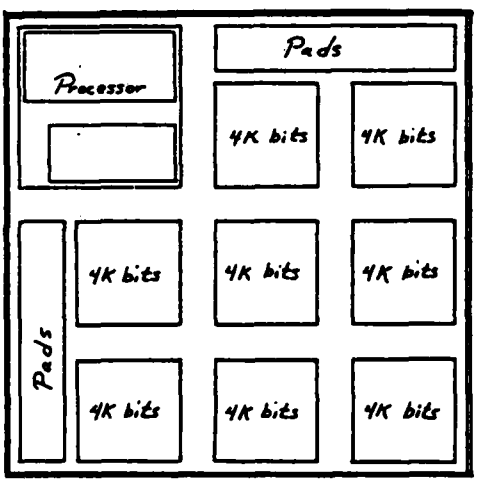


Figure 1: Mosaic Element Floorplan

The Mosaic element is synchronous, with 2-phase non-overlapping clocks supplied externally. The storage cycle, processor microcycle, and datapath operations occur in parallel in one clock cycle.

PROCESSOR ORGANIZATION

Figure 2 is a detailed block diagram of the processor, while figure 3 is the floorplan of the core of the processor, without the surrounding storage and pads. The processor has two principle components: a datapath/port block, and a controller. Each is a very dense, mostly metal-limited block of layout. The datapath/port block is functionally centered around the processor's single 16-bit internal data bus; it is controlled by signals issued by the PLA-based controller.

DATAPATH

The Mosaic datapath contains those parts of the processor that communicate over the internal data bus. These parts include sixteen general purpose registers, an ALU-shifter with associated flags, a memory address section, an interrupt counter, four input ports, four output ports, and an interface to the memory data bus and memory address bus. The ports are discussed in the following section. The functional blocks in the datapath are organized in a bit slice pattern, one bit of the bus running through each bit slice, with a bit slice pitch of 34 lambda. In the first clock phase of each cycle the bus is precharged and the ALU-shifter computes a new result. The second (last) clock phase is used for the bus transfer and the ALU carry chain precharge.

The ALU obtains operands from a pair of latches, called X and Y, that are loaded from the bus. The ALU result serves as input to the shifter, which uses pass gates to route correctly shifted data to the ALU-shifter output. The ALU is logically very similar to the ALU in the OM design[4], with a pair of function blocks and a precharged pass transistor carry chain. Although the ALU does not use carry lookahead, it is optimized to the extent that it is not in the critical timing path. An associated special purpose register, the Multiplier/Product, allows the processor to perform a multiply step in one microcycle. The multiply macroinstruction produces a 32-bit unsigned product in 20 microcycles.

The processor maintains four flags associated with the ALU/shifter. These are the familiar Z (zero result), N (negative result), V (two's complement overflow), and C (carry/not borrow). The controller cannot sense the values of the flags directly. Instead, a fixed 3-bit field in conditional branch macroinstructions specifies one of eight branch conditions. These three bits, as well as the values of the four flags, are inputs to a small PLA that produces one bit of output, the "flag condition". This bit is an input to the controller, which tests it when performing the conditional branch instructions. The branch condition codes were assigned carefully so the flag condition

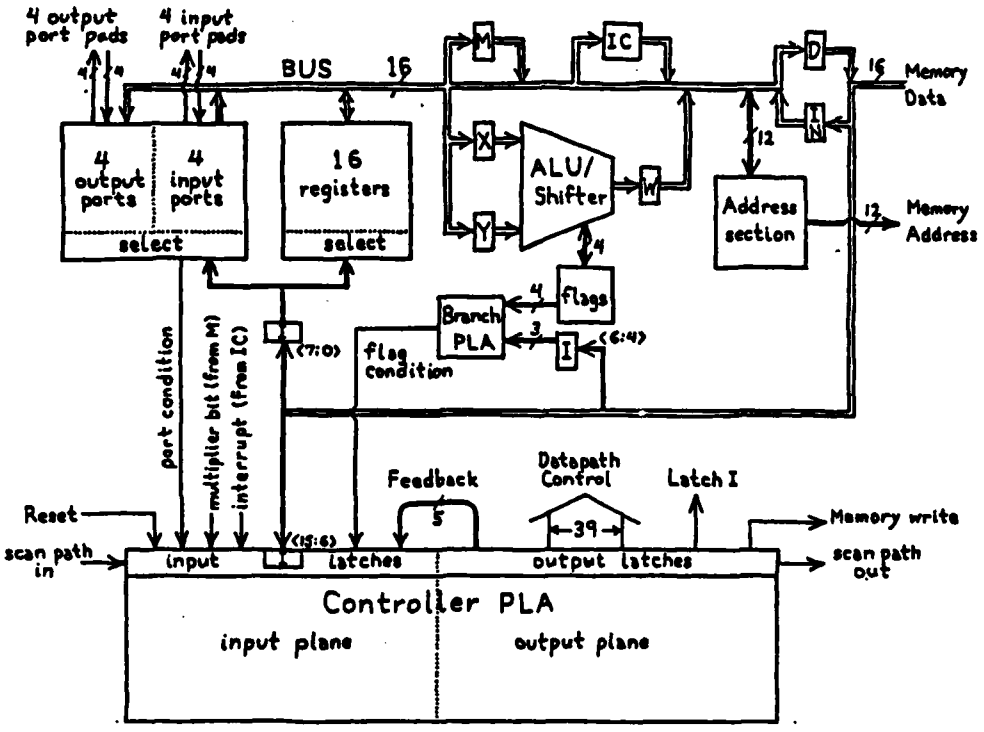


Figure 2: Processor Block Diagram

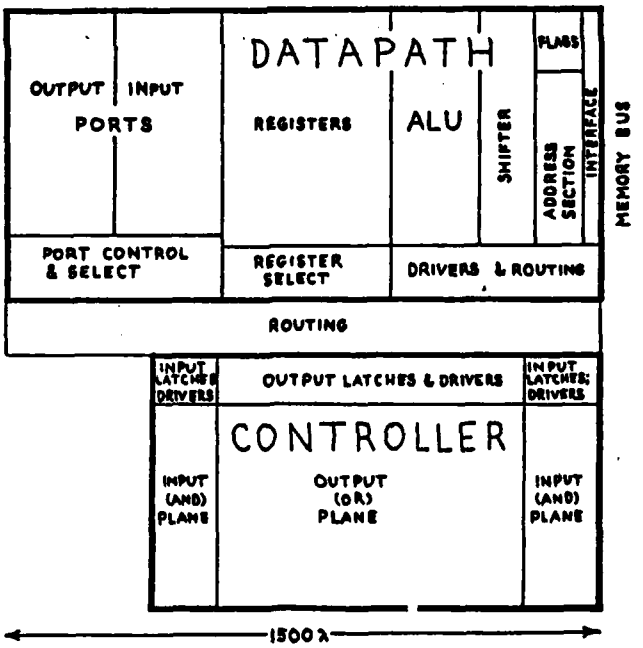


Figure 3: Processor Floorplan

PLA requires only seven implicants. Since the PLA is so small, it fits neatly next to the flags in the upper right corner of the processor, in a region formed by removing the top four bit slices of the address section.

On every microcycle the address section emits a new memory address onto the 12 memory address wires that come out of the right edge of the datapath. A 12-bit address is currently sufficient for the number of words of memory we can place on chip. The address generation section houses the program counter (PC) register, the current refresh address (RA) register, and an incrementer used with both registers. The controller guarantees that the RA is incremented and issued to the memory frequently enough, at least once every 8 microcycles, to keep the dynamic memory refreshed. Only memory cycles which would otherwise go to waste are used for refresh cycles.

The processor can generate timed interrupts using its interrupt counter (IC). The IC is a 16-bit register which counts down once per microcycle and causes an interrupt when it reaches zero. In order to guarantee interrupt service in bounded time, the port-wait states must be interruptible. Thus the side effects of any input or output instruction that cannot be completed immediately are reversed, and the instruction is refetched and restarted. Timed interrupts are useful for decoupling communications from processing, (*eg*, to implement automatic message routing, and to buffer large blocks of data) and to give the processor a sense of time (*eg*, for heuristic searches).

PORTS

Mosaic processors communicate with each other through their ports. Each processor has 4 input ports and 4 output ports. Connecting an output port of one processor to an input port of another processor forms a two-word fifo. Each output port is based on a parallel-in, serial-out shift register; each input port is based on a serial-in parallel-out shift register. The communication between input and output ports is bit serial.

Mosaic's implementation of the ports requires only a single wire, called the port link, to connect an input to an output port. When a port is not ready to perform a serial transfer, because it is an output port with no data or an input port with unremoved data, it clamps the port link to ground. On the microcycle when both ports are ready to perform a transfer, neither port grounds the port link and it is pulled up to VDD by an external pullup resistor. Both ports recognize this signal as the "start bit" of a transfer, much as in RS-232 data communications. The next 16 microcycles pass the data serially on the port link, and then the ports revert to the clamp-if-not-ready state.

This protocol allows multiple input ports to be connected to the same link: all input ports receive data from the output port beginning on the cycle when all the ports are ready. We didn't notice this feature until after we had running chips.

CONTROLLER

On every microcycle the Mosaic controller issues a new set of signals to control the datapath. The original plans for the controller assumed a rather conventional organization in which microcode words were fetched from a ROM, and a new ROM address was computed every microcycle by a conglomeration containing an incrementer, multiplexers, and other miscellaneous logic. This design was simplified when we realized that we could efficiently program a PLA to perform most of the original controller's function. An auxiliary PLA which controlled the ALU/shifter proved to be very troublesome because we could not find a placement for it that did not result in large wiring channels and expanses of white space. We finally eliminated the auxiliary PLA by learning to program the main controller PLA to perform its function. The controller became merely a PLA with latches.

In most microcircuit instruction processors the datapath is the more regular part, and the control the less regular part. In the Mosaic processor, the controller is even more regular than the datapath.

The controller has 20 inputs: 10 bits from the instruction register, the flag condition, the port condition, the multiplier bit from the multiplier/product register, the interrupt request from the interrupt counter, the processor reset, and 5 feedback bits (outputs from the controller clocked directly back to the controller input). The instruction register (I) holds the current macroinstruction and can be latched from the memory data bus on command from the controller. So little feedback state is needed because much of the state is held in the instruction register, and the sequences to implement macro instructions are short, typically 5 microcycles. (The shorter instructions are in practice executed more frequently, so

the average execution time is 4 microcycles.) Most of the 46 outputs from the controller go to clock-AND bootstrap drivers that drive control lines into the datapath. These outputs are effective during the microcycle following the microinstruction fetch.

INSTRUCTION SET

The tables in figure 4 summarize the macroinstruction set. All instructions are one word followed by zero, one, or two words of immediate data. In the first instruction word, the two 4-bit fields J and K can be used to specify one of the general registers. In some instructions, the K field may specify one of the ports or a branch condition instead.

There are two types of instructions: MOVEs and Arithmetics. MOVE instructions fetch an operand specified by a 3-bit MSOURCE field, and assign it as specified in the 4-bit MDEST field. The MOVEs incorporate subroutine linkage and branches: specifying an MDEST of PC performs a jump; an MDEST of PCF→Q——R; X→PC performs a subroutine call by pushing the current PC on a stack, and then assigning it.

Arithmetic instructions fetch two operands, X and Y, as specified by the 3-bit MODE field. (The X and Y operands in fact correspond to the hardware registers X and Y at the input to the ALU.) Then they perform the operation specified by the 4-bit OP field, which requires computing some function of X and Y, and usually assigning a result as specified by the MODE.

Instructions that write to an output port wait until there is room in the fifo. Instructions that read from an input port wait until there is a word to read, and can optionally "advance" the port (remove the word from the fifo).

The richness of this instruction set is justified by the code compactness it offers in its environment of scarce on-chip memory.

MICROCODE

The speed, simplicity, and compactness of this design owe much to the realisation that the controller need be nothing more than a PLA with latches. But a PLA is not merely sufficient; it is convenient and easy to program for an instruction set such as Mosaic's in which microinstruction sequences are short but heavily branched.

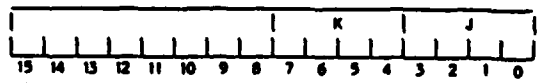
We chose to view each of the 120 implicants in the PLA as a word of microcode. More than one word of microcode can be active (that is, more than one implicant can be TRUE) in any given microcycle. Usually only one word is active at a time, but there are important exceptions. In these cases, the outputs are partitioned into disjoint sets, such that each active word has no TRUE outputs (transistors in the OR plane) outside its set. The effect of multiple active words used in this restricted manner is like that of multiple disjoint PLAs, but the physical layout retains the regularity of one PLA. In return for this self-imposed restriction, the absolute true/complemented sense of the individual outputs is irrelevant, the microcode assembler and assembly language is simpler, and the microcode is easier to understand.

A simple microcode assembler, written in SIMULA, reads the source microcode and assembles it into a runtime data structure. From here the assembler can output the code in any of several formats, including Earl source code. The assembler also contains an *ad hoc* register-transfer level simulator of the entire processor. This simulator served as an initial debugger for the processor design, and is still the initial testing ground for modifications in the processor and its microcode.

To illustrate some features of the microcode programming style and processor timing, the rest of this section is a blow-by-blow description of the execution of a sample macroinstruction. Figure 5 shows the assembly of the instruction ADD #7,R1,R2, the 4 microcode words required to execute it, and the behavior of various parts of the processor in the vicinity of these 3 cycles. This instruction adds immediate data 7 to the contents of register 1, and stores the result in register 2. The instruction executes in 3 microcycles, corresponding to the first, second, and last two microcode words (the last two are active simultaneously).

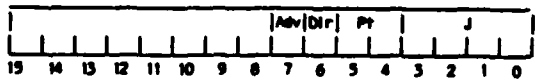
The tokens ".decode", ".get", and ".go" are mnemonics for feedback states; they appear both in the input conditions and in the next state outputs. The first microcode word, "DECODE:", is in fact the first word of every instruction. It becomes active any time the feedback state is ".decode", and no interrupt is pending ("Interrupt=0"). Previous microcode has ensured that a new macroinstruction was fetched on the previous cycle. Thus "DECODE:" latches it into

ALL INSTRUCTIONS:



followed by 0, 1, or 2 words of immediate value.

When K specifies a port:

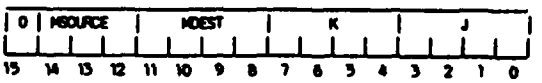


Pt is the port number; specifies one of 4 ports
Dir=0 for output port; Dir=1 for input port
Adv=1 to advance port if input port is read

KEY:
- Rn is register number n.
- Rn++ is register number n, incremented after reading.
- --Rn is register number n, decremented before reading.
- val is the immediate value.
- @z is the memory word whose address is z.
- OutPort is output port number Pt.
- InPort is input port number Pt.
- A | B means concatenation of bit field A and bit field B.
- f<i> means the i-th bit of f.
- f<i:j> means i-th to j-th bits of f.
- FPC is the flag/PC word: C | V | N | Z | PC

SPECIAL CASES: RESET: FPC -> @(-1); 0 -> PC
INTERRUPT: 0 -> IC; FPC -> @(-2); @(-3) -> PC

MOVE INSTRUCTIONS:



R means RK when MSOURCE is 0, 1, 2, or 3; R means RJ otherwise.

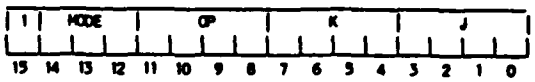
MSOURCE	X
0	RJ
1	@RJ
2	@RJ++
3	@(RJ+val)
4	val
5	@val
6	InPort
7	IC

MDEST	effect
0	X -> R
1	X -> @R
2	X -> @R++
3	X -> @(R+val)
4	X -> @--R
5	X -> @val
6	X -> OutPort
7	X -> IC
8	IC -> R; X-> IC
9	X -> PC
A	X -> FPC
B	X -> PC if flag condition K true
C	X -> PC if flag condition K false
D	X -> PC if port K is not ready
E	X -> PC if port K is ready
F	FPC -> @--R; X -> PC

FLAG CONDITIONS:

K	flag condition		K	flag condition	
0	V	(overflow)	4	Z	(zero)
1	N	(negative)	5	Z or N	(<= zero)
2	~C	(Carry = 0)	6	Z or ~C	(unsigned <=)
3	N xor V	(signed <)	7	Z or (N xor V)	(signed <=)

ARITHMETIC INSTRUCTIONS:



MODE	Dir	X	Y	Dest	Assembly language example
0		RJ	RK	RK	ADD R1,R2
1		val	RJ	RK	ADD #val,R1,R2
2		@RJ	RK	RK	ADD @R1,R2
3		@val	RJ	RK	ADD @#val,R1,R2
4	0	RJ	0	OutPort	MOV R1,P1
5	0	val	RJ	OutPort	ADD #val,R1,P1
4	1	InPort	RJ	RJ	ADD P1,R1 (ADD P1+,R1 to advance)
5	1	val	InPort	RJ	ADD #val,P1,R1 (ADD #val,P1+,R1 to advance)
6		@RJ	RK	@RJ	ADD M @R1,R2
7		@val	RJ	@val	ADD M @#val,R1

All Arithmetic instructions modify the Z, N, and V flags.

OP	Instruction	Assembler Mnemonic	Effect		Carry flag modified?
0	INCrement	INC	X + 1	-> Dest	no
1	DECrement	DEC	X - 1	-> Dest	no
2	Arithmetic Shift Right	ASR	X<15>|X	-> Dest|C	yes
3	ROtate Right	ROR	C|X	-> Dest|C	yes
4	ROtate Left	ROL	X + X + C	-> Dest	yes
5	Logical Shift Right	LSR	0|X	-> Dest|C	yes
6	Rotate Nibble Right	RNR	X<3:0> | X<15:4>	-> Dest	no
7	ADD	ADD	X + Y	-> Dest	yes
8	ADD with Carry	ADDC	X + Y + C	-> Dest	yes
9	SUBtract	SUB	Y - X	-> Dest	yes
A	bitwise COMplement	COM	~X	-> Dest	no
B	bitwise eXclusive OR	XOR	X exclusive or Y	-> Dest	no
C	bitwise AND	AND	X and Y	-> Dest	no
D	bitwise OR	OR	X or Y	-> Dest	no
E	COMpare	CMP	X - Y		yes
F	MULtiply	MUL	high word(X*Y)	-> RJ	no
			low word(X*Y)	-> RK	
			modify Z,N,V based on high word		

Figure 4: Diagram of the Complete Instruction Set

A macroinstruction, assembly language:

```
11: ADD #7,R1,R2
```

A macroinstruction, binary code:

```
10: ...                  [last word of previous instruction]
11: 1001 1001 0010 0001  [first word of instruction]
12: 0000 0000 0000 0111  [immediate value = 7]
13: ...                  [first word of next instruction]
14: ...                  [immediate value for next instruction, or
                             first word of instruction after next]
```

Source microcode for executing the macroinstruction:

The syntax for a source microcode word is:

```
word <mnemonic>: <inputs>                    :: <outputs>

word DECODE: .decode    Interrupt=0 ::  IN->I  RA++->A  RJ=> X Y M   .get

word #,J,K:  .get I= 1 0 0 1        ::  saveC  PC++->A  IN=> X       .go

word ADD:    .go I= 1 * * * 1 0 0 0 :: ALUONLY GP=86 Cin=0 NOshift setZNV setC
word ALU->K: .go I= 1 0 * *         :: NOALU   PC++->A    W=> RK     .decode
```

Processor timing in executing the macroinstruction:

micro-cycle number	microcode word(s) being fetched	microcode word(s) controlling processor	memory address being computed	memory address	memory data available	ALU function and bus transfer
	DECODE:	...	PC+1 = 12 (immediate value)	11 (ADD instr)	...	...
1	#,J,K:	DECODE:	RA+1 (new refresh address)	12 (immediate value)	ADD instr (latch into I register)	R1=> X,Y,M
2	ADD: and ALU->K:	#,J,K:	PC+1 = 13 (1st word of next instr)	(refresh address)	7 (immediate value)	7=>X
3	DECODE:	ADD: and ALU->K:	PC+1 = 14 (immed for next instr)	13 (1st word of next instr)	(refresh data)	X+Y->W W=>R2
	...	DECODE:	RA+1 (new refresh address)	14 (immed for next instr)	(1st word of next instr)	...

Figure 5: Example of a macroinstruction execution

the instruction register ("IN→I") at the start of the cycle. The controller has not had time to branch based on the new instruction, but the J and K fields will have arrived at the register decoder in time to select a register to drive the bus on that cycle; thus this microcode word fetches one of the registers to all of the destinations where it might be needed ("RJ⇒ X Y M"). This "register prefetch" saves a cycle in most instructions. It is too early to know what to do with the next memory cycle, so the microcode uses it as a refresh cycle ("RA++ →A", a macro for "RA→ INC Add1 INC→A A→RA").

The next microcode word "#,J,K:" is conditional on the first bit (1) and MODE bits (0 0 1) of the instruction register ("I= 1 0 0 1") and corresponds to an arithmetic instruction with an immediate value and a register as operands. In the complete microcode there is also a microcode sequence conditional upon each of the other possible MODE fields. The MODE in this example specifies operand X as an immediate value, which is obtained from the memory data bus via the memory data input buffer ("IN⇒ X"). The PC is then incremented past the immediate value ("PC++ →A") in order to begin fetching the next instruction. The next state ".go" indicates that all operands have been fetched and the code for the operative part of the instruction should take over.

The last two microcode words, "ADD:" and "ALU → K:" are active simultaneously and complete the macroinstruction. In the "ADD:" word the token "ALUONLY" indicates that this word specifies only ALU outputs (i.e. it has no transistors in the OR plane for other outputs) while "NOALU" in the "ALU→ K:" indicates that this word controls the rest of the outputs. The "ADD:" word instructs the ALU to add its inputs, X and Y, by specifying the appropriate Generate and Propagate codes ("GP= 86"), the carry-in ("Cin= 0"), and the type of shift ("NOshift"). The complete microcode contains similar words corresponding to the other arithmetic operations: subtract, increment, etc. They are independent of the MODE field of the instruction but dependent on the OP field ("I= 1 * * * 1 0 0 0", since OP code for ADD is 1000).

The "ALU→K:" word deposits the ALU output in register K ("W⇒ RK"). Other words, dependent on the MODE but independent of the OP code, handle the other possible destinations. Thus the orthogonality in the macroinstruction set, arithmetic OPs versus MODEs, is represented directly in the microcode. Note that only one microcode word, "ALU→K:", is needed to handle four MODE cases, since the MODEs have been carefully encoded so that one input condition ("I= 1 0 * *"), decodes all four cases. Careful encoding such as this throughout the instruction set has led to more compact microcode. In "ALU→K:" the PC is incremented and used as the memory address ("PC++ →A"), as it is in the last cycle of all instructions. This begins fetching the word after the next instruction, in case the next instruction takes an immediate value and needs to use it in its second cycle.

STORAGE

Although specialized semiconductor processes provide higher storage density than those suitable for the Mosaic processor, a processor with on-chip memory has many advantages over processor and memory in separate packages. These advantages include reduced volume, pin count, signal energy, and driver delay, resulting primarily from the integration of the memory bus into a single package.

For each storage bank, a two-bus three-transistor dynamic RAM memory cell is organized in 64x64 bits with a 16-bit word interface. All the banks operate in parallel to accomplish parallel refresh, and provide a read and pipelined write operation every processor microcycle (roughly 300 tau). Each memory access starts with a word-line access followed almost always by a refresh write to the same word-line on the next cycle. Because this write occurs in the same cycle as the next read, the storage control is somewhat subtle.

Mandating a write causes one of the 4 words read from the selected memory bank to be replaced with write data from the processor. This write data is written in the next cycle, in parallel with the next read. However, if the read is to the same word line as the pipelined write, stale data is accessed, and the subsequent write back to the word line would permanently store incorrect data. For this reason a write back is disabled on the second cycle after a write cycle. This form of pipelining imposes domain restrictions upon the microcode in that consecutive writes, refresh following write, and write followed by read to the same address will all fail. The first two conditions do not occur in the microcode, and the third condition occurs only by writing into the instruction stream.

CIRCUIT DESIGN

Some of the performance and layout simplicity of Mosaic is due to the simple clock-AND bootstrap driver shown in figure 6. It is used extensively and in several variations both in the processor and storage sections. In the processor, this clock-AND is used to produce control signals that are the logical-and of a PLA output and a clock. In the storage, the clock-AND is used so extensively in driving select and data lines that depletion transistors are completely absent.

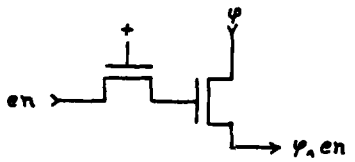


Figure 6: Clock-AND Circuit

Although referred to as a "driver," this clock-AND does not provide power amplification of the clock, but rather passes a replica of the "hot clock" input, whatever its HIGH voltage, to the output as gated by an enable signal of low energy. The clock signal typically switches between ground and 7 volts with VDD = 5 volts, but the chips also work correctly at reduced speed with 5 volt clocks. The delay and power dissipation of these clock-ANDs is almost negligible, and so the clock driving problem, together with the power dissipation usually required in control signal drivers, is exported to outside the chip where it can be dealt with using special driver circuits. This hot clock technique improves performance in pass structures, and also makes the performance much less sensitive to variations in the depletion threshold voltage than in conventional Mead-Conway designs. Precharge techniques are also used extensively in this chip, both to save power and for speed.

DESIGN TOOLS

The layout and verification was accomplished on a VAX-11/780 running Berkeley Unix with design tools written in MAINSAIL and C. Circuit design and optimization relied primarily on tau model calculations. SPICE was used to evaluate bootstrap effects, technology dependence, and critical timing paths.

The processor design is represented by 10,000 lines of code, interpreted by Earl[6], a constraint solving composition and geometry tool. Although the parts are composed in a rectangular bounding box discipline, the geometry internal to cells includes arbitrary angles and approximations of circular arcs, a form of "Boston geometry" that can be specified very easily in Earl. This unusual layout style is estimated to have reduced silicon area by 10% over 45-degree angle geometry, and by about 25% over Manhattan geometry.

For the design verification, the entire logic design was coded and simulated using the ternary switch level simulator MOSSIM[8] to verify logical correctness. After the layout was complete, raster extraction of layout using a Boston geometry circuit extractor produced a switch network that was used for MOSSIM II[9] simulations.

TESTING

First silicon for the Mosaic processor, received on 9 February 1983, 34 days after the CIF was submitted to MOSIS, was tested immediately and found to run code at a 7 MHz clock rate at room temperature. Subsequent processors fabricated using a faster process (still with a 4 micron feature size) ran at up to 11 MHz at room temperature.

Initial testing was accomplished by running the same code that had been used for switch level simulations. Subsequent testing using more extensive test programs discovered minor bugs that have been fixed in subsequent microcode. A scan path included in the original design between the datapath and controller was not used, although it might have been useful if anything had been seriously wrong.

Overall, our testing experiences have been quite similar to those reported by several other university groups, and point to two interesting development in testing for design verification. First, verification tools have advanced to the extent that nearly the entire design verification task is now accomplished before first silicon. Second, chips that are systems rather than components turn out to be simpler to test by placing them in their system environment than in a conventional tester, and the same tools that are used to program these systems serve to develop thorough tests of their function.

ACKNOWLEDGEMENTS

Chris Kingsley - Earl, Mike Schuster - Fsim, Howard Derby - early design, OM2 & GMP - ideas.

REFERENCES

[1] Sally A Browning, *Computations on a Tree of Processors*, Proceedings of the Caltech Conference on VLSI, January 1979, Computer Science, Caltech.

[2] Sally A Browning, *Hierarchically Organized Machines*, Section 8.4 in Mead & Conway[4].

[3] Sally A Browning and Charles L Seitz, *Communication in a Tree Machine*, Proceedings of the Second Caltech Conference on VLSI, January 1981, Computer Science, Caltech.

[4] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

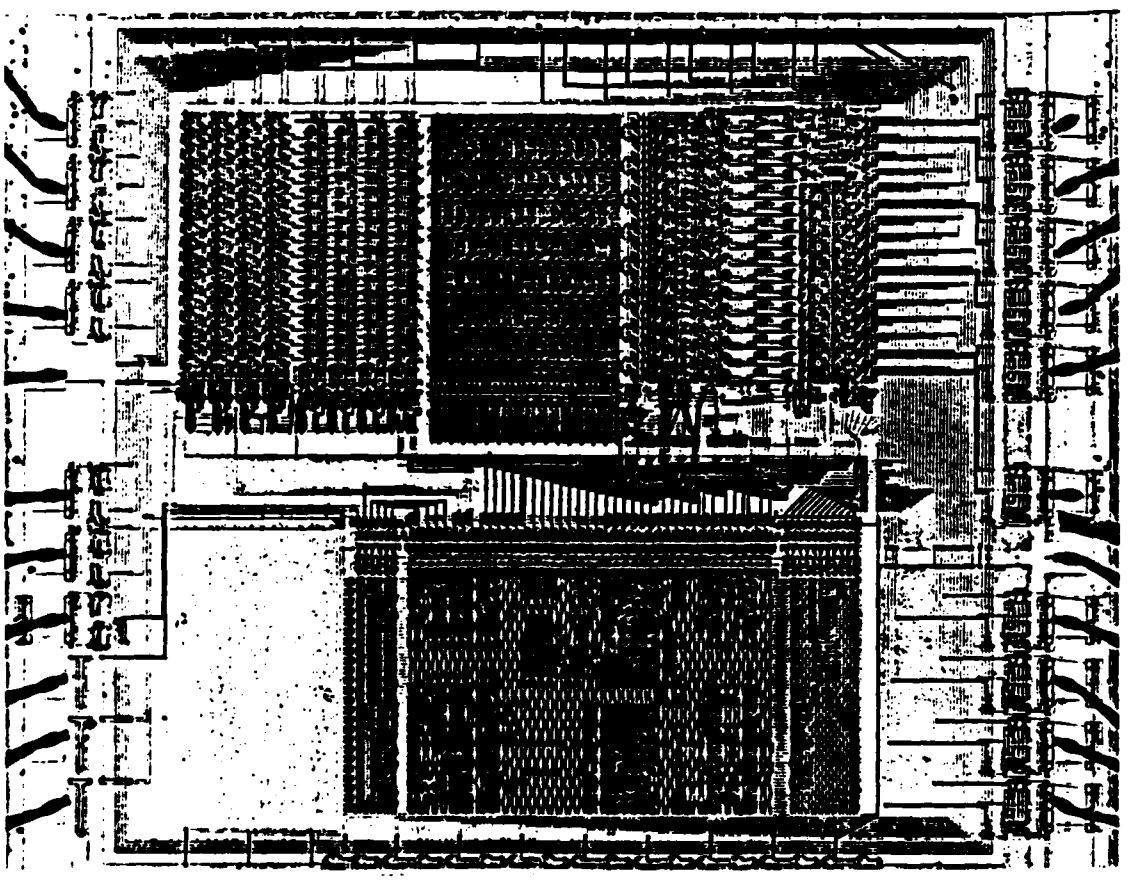
[5] Charles L Seitz, *Ensemble Architectures for VLSI*, Proceedings of the MIT Conference on Advanced Research in VLSI, January 1982, Artech Books, 1982.

[6] Chris Kingsley, *Earl: An Integrated Circuit Design Language*, Technical Report 5021, Computer Science, Caltech, June 1982.

[7] Charles L Seitz, *Experiments with VLSI Ensemble Machines*, Technical Report 5102, Computer Science, Caltech, October 1983.

[8] Randal E Bryant, *A Switch-Level Model and Simulator for MOS Digital Systems*, Technical Report 5065, Computer Science, Caltech, January 1983.

[9] R. Bryant, M. Schuster, D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual*, Technical Report 5033, Computer Science, Caltech, March 1982.



Prototype Mosaic Processor

CONCURRENT FAULT SIMULATION OF MOS DIGITAL CIRCUITS

Michael D. Schuster and Randal E. Bryant

California Institute of Technology
Pasadena, California 91125

5101:TM:83

To be presented at the Conference on Advanced Research in VLSI, to be held at the Massachusetts Institute of Technology, January 1984. Proceedings published by Artech House, Inc., Dedham, MA 02026.

ABSTRACT

The concurrent fault simulation technique is widely used to analyze the behavior of digital circuits in the presence of faults. We show how this technique can be applied to metal-oxide-semiconductor (MOS) digital circuits when modeled at the switch-level as a set of charge storage nodes connected by bidirectional transistor switches. The algorithm we present is capable of analyzing the behavior of a wide variety of MOS circuit failures, such as stuck-at-zero or stuck-at-one nodes, stuck-open or stuck-closed transistors, or resistive opens or shorts. We have implemented a fault simulator FMOSSIM based on this algorithm. The capabilities and the performance of this program demonstrate the advantages of combining switch-level and concurrent simulation techniques.

This research was supported in part by the IBM Corporation and by the Defense Advanced Research Contracts Agency, ARPA Order 3771. Michael Schuster was supported in part by a Bell Laboratories Ph.D. Scholarship.

CONCURRENT FAULT SIMULATION OF MOS DIGITAL CIRCUITS

Michael D. Schuster and Randal E. Bryant

Department of Computer Science
California Institute of Technology
Pasadena, California 91125

ABSTRACT

The concurrent fault simulation technique is widely used to analyze the behavior of digital circuits in the presence of faults. We show how this technique can be applied to metal-oxide-semiconductor (MOS) digital circuits when modeled at the switch-level as a set of charge storage nodes connected by bidirectional transistor switches. The algorithm we present is capable of analysing the behavior of a wide variety of MOS circuit failures, such as stuck-at-zero or stuck-at-one nodes, stuck-open or stuck-closed transistors, or resistive opens or shorts. We have implemented a fault simulator FMOSSIM based on this algorithm. The capabilities and the performance of this program demonstrate the advantages of combining switch-level and concurrent simulation techniques.

INTRODUCTION

Test engineers use fault simulators to determine how well a sequence of test patterns, when applied to the inputs of an integrated circuit, can distinguish a good chip from a defective one. The fault simulator is given a description of the good circuit, a set of hypothetical faults in the circuit, a specification of the observation points of the test (e.g. the output pins of the chip), and a sequence of test patterns. It then simulates how the good circuit and all of the faulty circuits would behave when the test patterns are applied to the inputs. A fault is considered detected if at any time the simulation of that particular faulty circuit produces, at some observation point, a logic value different than that produced by the good circuit. By keeping track of which faults have been detected and which have not, the fault simulator can determine the *fault coverage* of the test sequence, which is defined as the ratio of the number of faults detected to the total number simulated. The simulator can also provide the user with information about which faults have not been detected, either because the test sequence failed to exercise the defective part of the circuit, or because the sequence failed to make the effect of such an exercise visible at some observation point. This information guides the engineer in extending or modifying the test sequence to improve its fault coverage. Such a tool is invaluable for developing test patterns for today's complex digital systems.

For a large integrated circuit such as a microprocessor chip, thousands of faults must be simulated to adequately characterise the fault coverage of a test sequence. Furthermore, test sequences can involve thousands of patterns. Hence a simple *serial* simulation, in which the good circuit and each faulty circuit are simulated separately, would require far too much computation. Fortunately, clever algorithms can reduce the amount of computation considerably. A technique known as *concurrent simulation*[1] exploits the fact that a faulty circuit typically differs only slightly from the good circuit. Rather than simulating each circuit separately, only the good circuit is simulated in its entirety. The simulator keeps track of how the network state of each faulty circuit differs from the network state of the good circuit by selectively simulating portions of the faulty network. To the user, it appears as if the program is simulating many circuits concurrently, but the amount of CPU time required is a small factor (e.g. often less than 10 times) greater than the time required to simulate the good circuit alone. Furthermore, the simulator can easily determine when a faulty circuit produces a value different than the good circuit at some observation point without stor-

This research was supported in part by the IBM Corporation and by the Defense Advanced Research Contracts Agency, ARPA Order 3771. Michael Schuster was supported in part by a Bell Laboratories Ph.D. Scholarship.

ing the entire output history of the good circuit simulation. Once a fault has been detected, the simulation of this particular faulty circuit can be dropped, thereby reducing the amount of computation required for the remainder of the simulation. Typically, the faults that cause great differences from the behavior of the good circuit, and hence require the most computational effort, are detected quickly. Consequently, fault dropping greatly improves the overall performance of the simulator.

Most existing logic simulators model a digital circuit as a network of logic gates, in which each gate produces values on its outputs based on the values applied to its inputs, and possibly on the value of its internal state. Some of these simulators extend the simple Boolean gate model, in which only the value 0 or 1 is permitted on each input and output, with additional logic values and special types of gates to model circuit structures such as busses and pass transistors. These simulators are not suitable for modeling faults in MOS digital circuits for two reasons: First, many MOS circuit structures cannot be adequately modeled as a set of logic gates. Creating gate-level descriptions of pass transistor networks, dynamic memory elements, and precharged logic is at best tedious and inaccurate, and at worst impossible, even with extended gate models. The user must translate the logic design by hand into a form compatible with the simulator, and the resulting simulation is inherently biased toward the user's understanding of the functionality of the circuit. Second, logic gate simulators are especially poor at predicting the behavior of a MOS circuit in the presence of faults. Even simple logic gates can become seemingly complex sequential circuits when a fault such as an open-circuited transistor occurs.[2] As a result, fault simulators based on logic gates can model only a limited class of faults, such as the gate outputs and inputs stuck-at-zero or stuck-at-one. Faults such as short circuits across transistors and between wires, or open circuits in transistors or wires, are beyond their capability. Furthermore, even the modeling of stuck-at faults is limited in accuracy when the logic gate description is an artificial translation of the actual circuit structure.

To remedy these problems with logic gate simulators, we propose that fault simulations of MOS circuits be performed at the *switch level* with the transistor structure of the circuit represented explicitly, but with each transistor modeled in a highly idealized way. This approach has proved successful for logic simulation in programs such as MOSSIM[3] and MOSSIM II,[4] because properties such as the bidirectional nature of field-effect transistors and the charge storage capabilities of the nodes in a MOS circuit are modeled directly, rather than by some artificial translation into logic gates. Unlike the precise, but time-consuming algorithms used by circuit simulators, switch-level simulators model the circuit in a sufficiently simplified way that they operate at speeds comparable with conventional logic gate simulators. Furthermore, our switch-level logic model is well suited for modeling a variety of failures in MOS circuits in a reasonably realistic way, because many faults can be viewed as creating new switch-level networks which differ from the switch-level representation of the good circuit. Hence, while the switch-level model has proved successful for logic simulation, it seems especially attractive for fault simulation. Hayes[6] has proposed the Connector-Switch-Attenuator representation of logic circuits for modeling faults, and our switch-level model has essentially the same capabilities.

We have adapted the technique of concurrent simulation to implement a fault simulator for MOS circuits, where the problem is viewed as one of simulating a large number of nearly identical switch-level networks. This program FMOSSIM can simulate a large variety of MOS circuits, under a variety of fault conditions, at much higher speeds than would be possible with serial simulation. Other concurrent fault simulators for MOS have been implemented,[5] but these could only model a very limited class of networks. In this paper, we will present an overview of the switch-level model and how different faults can be represented in it. We also discuss our concurrent, switch-level simulation algorithm and present performance results from FMOSSIM.

NETWORK MODEL

The following network model is implemented in the simulators MOSSIM II and FMOSSIM. It provides a more general transistor model than provided by other switch-level simulators, giving bet-

ter capabilities for fault injection. A switch-level network consists of a set of *nodes* connected by a set of *transistors*. Each node has a state 0, 1, or X, where 0 and 1 represent low and high voltages, respectively. The X state represents an indeterminate voltage arising from an uninitialized node, from a short circuit, or from improper charge sharing. No restrictions are placed on how transistors are interconnected.

Each node is classified as either an *input* node or a *storage* node. An input node provides a strong signal to the network, as does a voltage source in an electrical circuit. Its state is not affected by the actions of the network. Examples include the power and ground nodes *Vdd* and *Gnd*, which act as constant sources of the states 1 and 0, respectively, as well as any clock or data inputs.

The state of a storage node is determined by the operation of the network. Much like a capacitor in an electrical circuit, a storage node holds its state in the absence of connections to input nodes. To provide a simple model of charge sharing, each storage node is assigned a discrete *size* from the set $\{\kappa_1, \kappa_2, \ldots, \kappa_q\}$, where the sizes are ordered $\kappa_1 < \kappa_2 < \cdots < \kappa_q$. A larger storage node is assumed to have much greater capacitance than a smaller one. When a set of storage nodes charge share, the states of the largest nodes in the set override the states of the smaller nodes. The number of different sizes required (q) depends on the circuit to be simulated. Most circuits can be represented with just two node sizes. In this representation, high capacitance nodes such as busses assigned size κ_2, and all other nodes are assigned size κ_1.

A transistor is a device with terminals labeled *gate*, *source*, and *drain*. No distinction is made between the source and drain connections — each transistor is symmetric and bidirectional. Because transistors can be either *n-type*, *p-type*, or *d-type*, both nMOS and CMOS circuits can be modeled. A d-type transistor corresponds to a negative threshold depletion mode device. A transistor acts as a resistive switch connecting or disconnecting its source and drain nodes according to its type and the state of its gate node, as shown in Figure 1. Transistor states 0 and 1 represent open (nonconducting) and closed (fully conducting) conditions, respectively. The X state represents an indeterminate condition between open and closed, inclusive.

gate state	n-type	p-type	d-type
0	0	1	1
1	1	0	1
X	X	X	1

Figure 1. Transistor state function

To model the behavior of ratioed circuits, each transistor is assigned a discrete *strength* from the set $\{\gamma_1, \gamma_2, \ldots, \gamma_p\}$, where strengths are ordered $\gamma_1 < \gamma_2 < \cdots < \gamma_p$. A stronger transistor is assumed to have much greater conductance than a weaker one. When a storage node is connected to a set of input nodes by paths of conducting transistors, its resulting state depends only on the states of the input nodes connected by paths of greatest strength. The strength of a path is defined to equal the strength of the weakest transistor in the path. The total number of strengths required (p) depends on the circuit to be modeled. Most CMOS circuits do not utilize ratioed logic and hence can be modeled with just one transistor strength. Most nMOS circuits require only two strengths, with pull-up loads assigned strength γ_1 and all other transistors assigned strength γ_2.

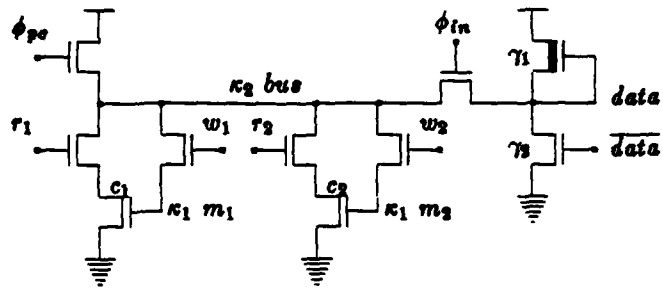


Figure 2. Three transistor dynamic RAM

As an example of a switch-level network, consider the three transistor dynamic RAM circuit shown in Figure 2. The bus node has size κ_2 to indicate that it can supply its state to the size κ_1 storage node (m_1 or m_2) of the selected memory element during a write operation (when w_1 or w_2 is 1) and to the size κ_1 drain node (c_1 or c_2) of the selected storage transistor during a read operation (when r_1 or r_2 is 1). The d-type pull-up transistor in the input inverter has strength γ_1, to indicate that it can drive the bus high only when the

strength γ_2 pull-down transistor is not conducting. The strengths of all other transistors in the circuit are arbitrary, since they are not involved in ratioed path formation (except possibly when faults are present).

The switch-level network model strikes a reasonable balance between a detailed electrical model and an abstract logical model. As a result of this abstraction, the model may not predict the true behavior of circuits such as sense amplifiers and arbiters which rely on detailed analog properties. Moreover, the network model does not contain enough detail to accurately model timing behavior, because even in circuits with straightforward logical behavior, timing can be subtle. However, experience has shown that switch-level simulation works quite well for verifying logic designs.

FAULT INJECTION

Faults are represented in FMOSSIM as though extra *fault transistors* were added to the network, much like that proposed by Lightner and Hachtel.[7] In the implementation, however, many of these faults are injected without actually adding fault transistors; nevertheless, the behavior is equivalent to what is described below. The gate nodes of the fault transistors are considered to be extra *fault inputs* to the network that control the presence or absence of the failures. A variety of MOS failures can be modeled with this method. For example, a short circuit between two nodes is modeled by connecting the nodes with a fault transistor that is open in the good circuit and closed in the faulty circuit. Similarly, an open circuit is modeled by splitting a node into two parts and connecting the resulting nodes with a fault transistor that is closed in the good circuit and open in the faulty circuit. By adjusting the strength of the fault transistor, the resistance of the short or open may be modeled in an approximate way. For example, if the strength of the fault transistor is set to γ_{p+1} (i.e. a strength greater than that of any normal transistor), then setting this transistor state to 1 shorts the source and drain nodes together such that they act as a single node. Moreover, because the state of each fault transistor can be controlled independently, both single and multiple faults can be injected.

Figure 3 illustrates the use of fault transistors

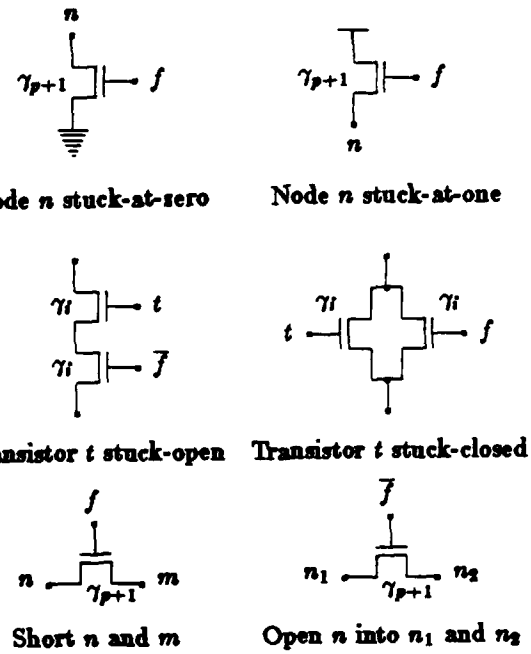


Figure 3. Modeling MOS failures

to create a variety of circuit faults. Those transistors with gate nodes labeled f are normally 0, but are set to 1 to create the fault; the transistors with gate nodes labeled $\bar{f}$ are normally 1, but are set to 0 to create the fault. A stuck-at-zero or stuck-at-one node fault can be modeled by inserting a strength γ_{p+1} fault transistor to short the node to *Gnd* or to *Vdd*, respectively. A stuck-closed transistor fault is injected by shorting the transistor's source and drain together with a fault transistor whose strength equals that of the failing transistor. Similarly, a stuck-open transistor fault is modeled by putting a fault transistor in series with it. In FMOSSIM, both stuck-at node states and stuck-at transistor states are implemented without extra fault transistors, while other faults require additional transistors to be inserted into the network.

Although FMOSSIM can model a larger class of faults than can be modeled by logic gate fault simulators, it still provides only a simplified representation of the faulty circuit. For example, the effects of manufacturing defects such as incorrect transistor thresholds, pinholes in the gate oxides, and variations in the circuit delays, cannot be described accurately. The effects of resistive

shorts and opens can only be approximated. In fact, even existing circuit simulators cannot model defects that change the basic nature of the devices, such as pinholes in the gate oxides. However, even if the fault models supported by our simulator do not exactly match the failure modes in actual chips, the program can still help the designer in developing a set of test patterns. For circuits implemented in bipolar technologies such as TTL, experience has shown that a test sequence that yields a high level of coverage for single stuck-at-zero and stuck-at-one faults in the logic gate network generally provides a good test of the circuit. It seems reasonable to expect that the test coverage measured by a switch-level fault simulator for an idealized set of faults should reliably predict how well the test sequence will work on a MOS circuit. Such a conjecture, however, can only be confirmed by actual experience in a manufacturing environment.

Many faults in our model have the effect of creating an X state on a node when the good circuit has a 0 or 1. For example, if the control signal w_1 in the circuit shown in Figure 2 is stuck-at-zero, bit m_1 of the memory will never be initialized and will remain at X. On the other hand, if the precharge clock ϕ_{pc} is stuck-at-one, any time we try to read a 1 value out of a memory cell, a short circuit will develop between *Vdd* and *Gnd* giving an X on the bus. Whether or not such X's would be detected in an actual test depends on detailed characteristics of the circuit that cannot be predicted at the switch-level, such as the initial voltages of dynamic nodes, how the voltage would divide across a shorting path, and the thresholds of the devices sensing these X values. On one hand, a pessimist might argue that an X in a faulty circuit should be considered undetectable, because there is no guarantee that the X will produce an effect different than the state of the node in the good circuit. On the other hand, a fault that prevents the circuit from being initialized, such as a stuck-at-zero clock line, would clearly be quickly detected. As a compromise FMOSSIM allows the user to specify a *soft detect* limit l such that if in the good circuit some output changes both to 1 and to 0 at least l times each, while the output in a faulty circuit remains at X, then this fault is considered detected. This approach seems to work reasonably well in practice.

BEHAVIORAL MODEL

The operation of a MOS circuit is characterized in the switch-level model in terms of its *steady state response function*,[8,9] which can best be explained in terms of an analogy to electrical networks. A MOS transistor behaves as a voltage-controlled, nonlinear resistor where the voltages of its gate, source and drain nodes control the resistance between its source and drain. Suppose in a transistor circuit we could control the transistor resistances independently of the node voltages. For a given setting of the transistor resistances, such a circuit acts as a network of passive elements which, for a given set of initial node voltages, has a unique set of steady state node voltages. Thus a function that maps transistor resistances and initial node voltages to steady state node voltages gives a partial characterization of the behavior of a transistor circuit. The steady state response function provides just this sort of characterization, but in terms of node and transistor states 0, 1, and X. That is, for a given set of initial node and transistor states, the steady state response function yields the set of states which the storage nodes would eventually reach if all transistors were held fixed in their initial states. This function only approximates network behavior, since it does not describe the rate at which nodes approach their steady states nor the effects of the changing transistor states as their gate nodes change state.

In general, a switch-level network may contain nodes and transistors in the X state. Such states arise from improper charge sharing or (transient) short circuits even in properly designed networks. The behavior of a network in the presence of X states must be described in a way that is neither overly optimistic (i.e. ignoring possible error conditions), nor overly pessimistic (i.e. spreading X's beyond the region of indeterminate behavior). This can be accomplished by defining the steady state response of a node to be 0 or 1 if and only if the node would have this unique state regardless of whether each node and transistor in the X state had state 0 or 1; otherwise, the steady state of the node is defined to be X. Rather than computing the steady state for all possible combinations of the nodes and transistors in the X state set to 0 or 1 (a task of exponential complexity), an equivalent two-pass linear time algo-

rithm is used.[9] Each pass involves solving a set of equations expressed in a simple, discrete algebra using a relaxation algorithm.

Given a technique for computing the steady state response function, a switch-level logic simulator can be implemented that simulates the operation of the network by repeatedly performing *unit steps* until a stable state is reached. Each unit step involves computing the steady state response of the network, setting the storage nodes to these values, and setting the transistors according to the states of their gate nodes. This simulation technique implements a timing model in which transistors switch one time unit (i.e. one evaluation of the steady state response function) after their gate nodes change state. Such a timing model tells little about the speed of a circuit but usually suffices to describe the circuit's logical behavior. As with other unit delay simulations, this computation may not reach a stable condition due to oscillations in the circuit, and hence an upper bound must be placed on the number of steps simulated.

On a given unit step, often only a small portion of the network changes state, while the rest of the network remains inactive. Most logic simulators exploit this property by recomputing the output of a logic gate only if at least one of the gate's inputs has changed state. A similar effect is achieved in switch-level networks by viewing network activity as creating small *perturbations* of the network state, and only computing the effects of these perturbations incrementally rather than recomputing the state of the entire network. We say that a storage node is *perturbed* if it is the source or drain of a transistor that has changed state, or if it is connected by a transistor in the 1 or X state to an input node that has changed state. Such a perturbation can only affect storage nodes in the *vicinity* of the perturbed node, where two nodes are in the same vicinity if and only if there exists some path of transistors in the 1 or X state between the nodes which does not pass through any input nodes. This definition exploits the *dynamic* locality in the network where the source and drain of a transistor in the 0 state are considered to be electrically isolated. Typically, a vicinity contains only a few nodes, and hence activity remains highly localised.

```
unit-step(P);
    U:=∅;
    for each n ∈ P do
        U:=U ∪ update-vicinity(n);
    P:=∅;
    for each n ∈ U do
        begin
        P:=P ∪ perturb-transistors(n);
        update-node(n);
        end;
    return(P);
end unit-step;
```

Figure 4. Implementation of unit step

Figure 4 shows a simplified implementation of the unit step operation that uses this incremental perturbation technique to recompute only selected parts of the network state. The argument P is a set of perturbed storage nodes derived from either new data and clock inputs to the circuit or from the last unit step. For each of these nodes, *update-vicinity* finds all the nodes in the same vicinity, computes their steady state response, and returns a set of nodes that changed state. These updated nodes are accumulated in the set U. Vicinities are found by a *depth first search*[10] originating at the perturbed node and tracing outward through transistors in the 1 or X state from source to drain until an input node is encountered. As each node is added to the vicinity, it is flagged to avoid duplication and endless cycles. For each updated node n in U, *perturb-transistors* finds all transistors whose gate node is n and checks to see if they have changed state. Nodes perturbed by these changing transistor states are accumulated in a new set P in preparation for the next unit step. Finally, *update-node* sets each updated node to its new state.

CONCURRENT SIMULATION

We have seen that the presence or absence of a fault in a switch-level network is controlled by the state of a fault input node. Suppose the test patterns that specify data and clock input values are extended to include values for the network's fault input nodes. Then the behavior of a set of faulty circuits can be determined by repeatedly simulating patterns that differ only in selected fault input values. Hence, concurrent fault simulation can be viewed as the problem of efficiently applying a

large number of nearly identical test sequences to a single network. This viewpoint separates issues of fault modeling from concurrent simulation. For example, since values for fault input nodes are specified on an individual pattern by pattern basis, multiple and intermittent faults are easily modeled without changing the basic simulation algorithm. Furthermore, there are no inherent restrictions requiring that the data inputs of all test sequences be identical. Thus, concurrent simulation is useful not only for simulating faults, but for simulating sets of similar test patterns on a fault-free circuit.

The concurrent simulation algorithm is given a description of the network and a set of *test sequences* $T = \{t_0, \ldots, t_n\}$. A test sequence $t_i \in T$ consists of a sequence of test patterns, each specifying values for the data, clock and fault inputs of the network. The algorithm simulates the network to determine how each node behaves for each test sequence t_i. That is, at any point during the simulation, each node's state s_i in test sequence t_i is found. Since we assume that the behavior of the network differs only slightly from test sequence to test sequence, $s_i = s_0$ for most nodes in the network. This observation is exploited by representing node states compactly as a set of pairs $S = \{\langle t_i, s_i \rangle\}$, called a *state set*, where $\langle t_i, s_i \rangle \in S$ if and only if $i = 0$ or $s_i \neq s_0$. The behavior of the network for test sequence t_0 serves as a reference point, since states are explicitly stored only for test sequence t_0 and those sequences t_i whose states differ from t_0. For this reason, test sequence t_0 is called the *reference sequence*. For fault simulation, the reference sequence corresponds to the good circuit, while test sequences $t_i, i \neq 0$ differ only in selected fault input values, and hence correspond to faulty circuits. A node is said to be *diverged for* t_i if $s_i \neq s_0$. A node is said to be *diverged* if it is diverged for any t_i. If the gate node of a transistor is diverged, then the transistor itself is said to be diverged.

If a node is perturbed due to an input node or transistor changing state for the reference sequence, it is likely that the node is also perturbed for most other test sequences t_i. We exploit this observation by maintaining a set of perturbations of the form $P = \{\langle n_j, t_i \rangle\}$, called the *perturbation set*, where $\langle n_j, t_0 \rangle \in P$ if and only if node n_j is perturbed for the reference sequence t_0 and $\langle n_j, t_i \rangle \in P, i \neq 0$, if and only if n_j is perturbed for t_i but not for the reference sequence. The perturbation $\langle n_j, t_i \rangle \in P$, where $i \neq 0$, indicates that the network has behaved differently in the area near node n_j for test sequence t_i when compared to its behavior for the reference sequence.

As described above, each unit step of the conventional switch-level simulation algorithm computes a steady state response for each node in the vicinity of a perturbed node, updates those nodes that have new steady states, and returns a set of perturbations for the next unit step. To generalize this operation for concurrent simulation, observe that the perturbation $\langle n_j, t_0 \rangle \in P$ represents a perturbation not only for the reference sequence, but likely for most other test sequences. In general, the steady state response of nodes in a vicinity is a function of both their initial states as well as the states of the transistors whose source or drain node is in the vicinity. Thus, when the steady state response is computed for the nodes in some vicinity as a result of a perturbation for the reference sequence, we must check to see if any of the nodes or transistors are diverged. We expect that most of the time, for most test sequences t_i, nodes within the vicinity will not be diverged for t_i. In this case, the steady state response computation performed for the reference sequence will be valid for t_i, and hence there is no need to duplicate this computation for t_i. However, if some node n_j within the vicinity is diverged for t_i, then the steady state response computation using the states of the nodes and transistors for the reference sequence may not be valid for t_i. To guarantee that an accurate computation be performed for t_i, the perturbation $\langle n_j, t_i \rangle$ is added to P. In effect, we are simply scheduling a steady state response computation that will be performed sometime later. Diverged transistors are handled in a similar manner, for if some transistor with source n_s or drain n_d in the vicinity is found to be diverged for t_i, then the perturbations $\langle n_s, t_i \rangle$ and $\langle n_d, t_i \rangle$ are added to P.

To determine the steady state response for nodes in the vicinity of a perturbation $\langle n_j, t_i \rangle$, where $i \neq 0$, states of the nodes and transistors for test sequence t_i must be found. This involves searching node state sets S for elements of the form $\langle s_i, t_i \rangle$. If such an element is not found, then

$$Vdd = m_1 = m_2 = \overline{data} = \{\langle t_0, 1\rangle\}$$
$$Gnd = \phi_{pc} = \phi_{in} = data = \{\langle t_0, 0\rangle\}$$
$$r_2 = w_1 = w_2 = c_1 = c_2 = \{\langle t_0, 0\rangle\}$$
$$bus = \{\langle t_0, 1\rangle, \langle t_1, 0\rangle\}$$
$$r_1 = f_1 = \{\langle t_0, 0\rangle, \langle t_1, 1\rangle\}$$
$$f_2 = \{\langle t_0, 0\rangle, \langle t_2, 1\rangle\}$$

Figure 5. Initial Node States

the state for the reference sequence is used. To reduce search time, elements in both the state sets S and perturbation set P are kept sorted by test sequence.

As an example of this simulation technique, consider the circuit shown in Figure 2. An operation that sets node m_2 to 0 will be described. Suppose initially that nodes Vdd, m_1, m_2, *bus*, and $\overline{data}$ have state 1 and all other nodes have state 0. Two fault transistors are added to the network, one connecting node m_2 to Vdd whose gate is fault input f_1, the other connecting node r_1 to Vdd whose gate is f_2. For the reference sequence, both of these fault input nodes have state 0 so that the faults are absent. For test sequence t_1, f_1 has state 1 to inject fault r_1 stuck-at-one. For test sequence t_2, f_2 has state 1 to inject fault m_2 stuck-at-one. Due to the fault injected by t_1, *bus* and *Gnd* are connected by conducting transistors, hence *bus* is initially 0 for t_1. The representation of these initial states is shown in Figure 5.

To set m_2 to 0, nodes ϕ_{in} and w_2 must be set to 1. These changes perturb *bus*, *data*, and m_2, since they are connected to the source or drain of transistors that have changed state. The vicinity for each of these perturbed nodes contains *bus*, *data*, m_2, Vdd, and *Gnd*, and so their steady state responses are determined. All three storage nodes have steady states 0 due to the connection to *Gnd* through transistors whose gates are ϕ_{in} and $\overline{data}$. The states of Vdd and *Gnd* are unchanged since they are input nodes. Notice that the pull-up connection between *data* and Vdd has no effect on the steady state of *data* since the strength of this connection, which is γ_1, is less than the strength γ_2 pull-down connection between *data* and *Gnd*.

The steady state computation just described was performed relative to the reference sequence, since node states for the reference sequence were

$$Vdd = m_1 = \phi_{in} = w_2 = \overline{data} = \{\langle t_0, 1\rangle\}$$
$$Gnd = \phi_{pc} = data = \{\langle t_0, 0\rangle\}$$
$$r_2 = w_1 = c_1 = c_2 = \{\langle t_0, 0\rangle\}$$
$$bus = \{\langle t_0, 0\rangle, \langle t_2, \mathbf{X}\rangle\}$$
$$r_1 = f_1 = \{\langle t_0, 0\rangle, \langle t_1, 1\rangle\}$$
$$m_2 = f_2 = \{\langle t_0, 0\rangle, \langle t_2, 1\rangle\}$$

Figure 6. Final Node States

used to determine which nodes were within the vicinity as well as their steady state responses. This computation may be invalid for sequences t_1 or t_2 since *bus* has state 0 for t_2 and m_2 is connected to a conducting fault transistor for t_1. So that the appropriate steady state response computations will be performed for both t_1 and t_2, the perturbations $\langle bus, t_1\rangle$ and $\langle m_2, t_2\rangle$ are generated as the vicinity is found.

Consider the effects of perturbation $\langle m_2, t_2\rangle$. A vicinity containing *bus*, *data*, m_2, Vdd, and *Gnd* is found, as in the simulation for the reference sequence. The steady state response of *bus* depends on the strengths of the transistors whose gates are ϕ_{in} and w_2. If both of these transistors have strength γ_1, *data* stays 0 but *bus* becomes X due to the short between *Gnd* and Vdd through the fault transistor connected to m_2.

Now consider the effects of the perturbation $\langle bus, t_1\rangle$. In this case, the vicinity contains node c_1, in addition to those found above. The short between *bus* and *Gnd* has no effect, and c_1, m_2, *bus*, and *data* all have steady state responses equal to those in the good circuit. The representation of the final node states is shown in Figure 6.

In this example, we have seen that faults may affect the steady state response of nodes as well as which nodes are contained within a vicinity. By explicitly generating perturbations for diverged nodes and transistors when a vicinity in the good circuit is simulated, we exploit the locality of activity in each faulty circuit independent of activity in other circuits. Furthermore, this technique selectively simulates only differing portions of a faulty circuit, and hence simulation proceeds quickly.

PERFORMANCE RESULTS

As a test case for evaluating the performance of FMOSSIM, we simulated a 64 bit dynamic RAM

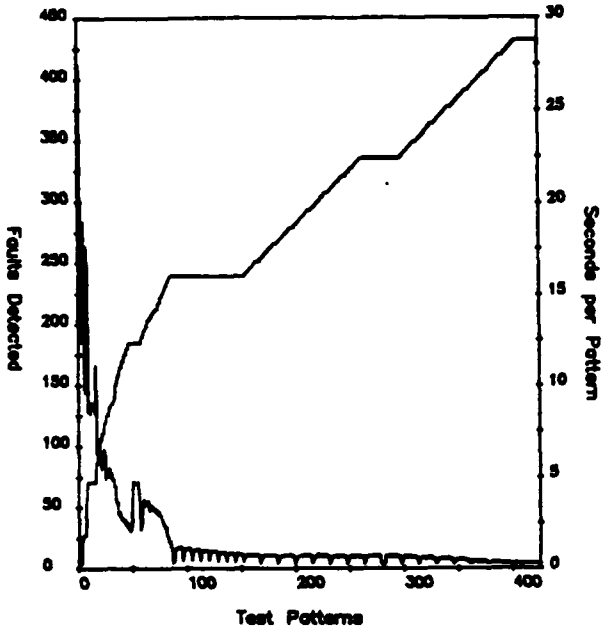


Figure 7. Performance on Memory Circuit

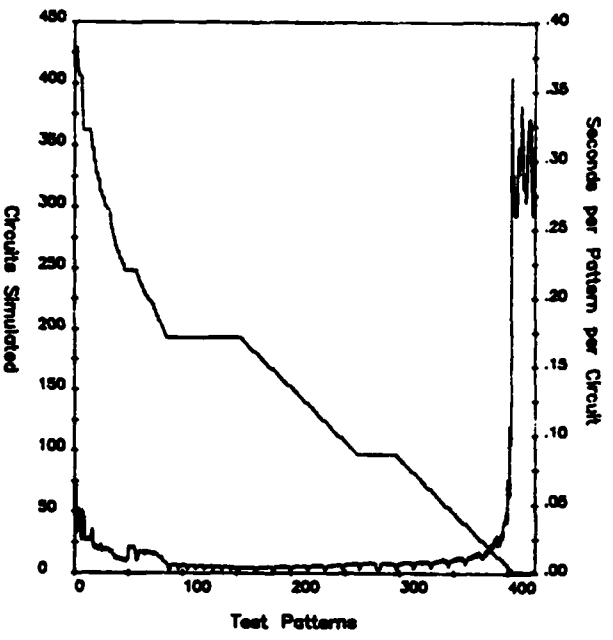


Figure 8. Effective Concurrency

circuit containing 374 transistors. This circuit incorporates a variety of MOS structures such as logic gates, bidirectional pass transistors, dynamic latches, precharged busses, and three-transistor dynamic memory elements. The circuit was simulated with 428 faults — each storage node stuck-at-zero, each storage node stuck-at-one, and pairs of adjacent busses shorted together. To validate the program, we also simulated other faults, including stuck-open and stuck-closed transistors. The simulator was implemented in the Mainsail programming language,[11] and run on a DEC-20/60.

Figure 7 illustrates the performance of FMOSSIM when simulating a test sequence consisting of a marching test[12] of the memory, together with special tests for the control logic. The curve climbing diagonally upward indicates the total number of faults detected as the test progresses. All faults were detected after 407 patterns. The falling curve indicates the CPU time required to simulate each pattern. This time starts at 27 seconds when the circuits are initialized. After 100 patterns, it drops to around 1 second as faults were detected and the simulations of these circuits were dropped. This time finally reaching 0.3 seconds at the end of the simulation, when only the good circuit is being simulated.

Figure 8 illustrates the performance advantage of concurrent simulation over simulating each faulty circuit separately. The curve falling diagonally to the right indicates the number of circuits being simulated as the test proceeds. The other curve indicates the CPU time required to simulate each pattern divided by the number of circuits being simulated for that pattern. This curve starts at about 0.05 seconds per pattern, drops to a low of 0.005 seconds once those faults causing major differences from the good circuit are dropped, and finally climbs back to 0.3 seconds when only the good circuit is being simulated. Considering that simulating a single circuit requires about 0.3 seconds per pattern, the effective benefit of simulating all of the circuits concurrently starts at 6 times serial simulation, rises to 60 times, and drops back down to 1.

Over the entire test sequence, simulating the good machine alone requires 2.5 CPU minutes. Our fault simulation requires 11 CPU minutes, whereas simulating each faulty circuit serially until it produces a different result than the good circuit would take almost 6 hours. Thus, in this case, concurrent simulation has a thirty-fold net advantage over serial simulation. Such a performance gain is clearly worth the effort.

CONCLUSION

Our experience with FMOSSIM has shown that it is a very useful tool for developing test sequences. Even when developing a test for a small section of an integrated circuit (such as an ALU or a register array), the fault simulator provides information that is hard to obtain by any other means. It quickly directs the designer to those areas of the circuit that require further tests. For example, in developing test sequences for the memory design described previously, we discovered that a simple marching test provided high coverage in the memory array itself, but that testing the control logic and peripheral circuits such as the input and output latches was more difficult.

It remains to be seen how the performance characteristics of FMOSSIM will vary as the size of the circuit and the number of faults to be simulated grows large. Even if it becomes impractical to run full-chip fault simulations with large numbers of faults, the program could still produce useful results by simulating portions of the chip, by eliminating faults that produce effects identical to other faults, or by simulating only a subset of the possible faults selected at random.

REFERENCES

[1] E. Ulrich, T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," *Design Automation Workshop Proc.*, June 19-73, pp. 145–160, and *IEEE Computer*, April 1974, pp. 39–44.

[2] R. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *Bell System Technical Journal*, Vol. 57, May-June 1978, pp. 1449–1473.

[3] R. Bryant, "MOSSIM: A Switch-Level Simulator for MOS LSI," *18th Design Automation Conference Proceedings*, July 1981, pp. 786–790.

[4] R. Bryant, M. Schuster, D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual*, Technical Report 5033, Department of Computer Science, California Institute of Technology, March 1982.

[5] J. Hayes, "A Fault Simulation Methodology for VLSI," *19th Design Automation Conference Proceedings*, July 1982, pp. 393–399.

[6] A. Bose, *et al*, "A Fault Simulator for MOS LSI Circuits," *19th Design Automation Conference Proceedings*, July 1982, pp. 400–409.

[7] M. Lightner, G. Hachtel, "Implication Algorithms for MOS Switch Level Functional Macromodeling, Implication and Testing," *19th Design Automation Conference Proceedings*, July 1982, pp. 691–698.

[8] R. Bryant, "A Switch-Level Model of MOS Logic Circuits," in J. Gray, ed., *VLSI 81*, Academic Press, August 1982, pp. 329–340.

[9] R. Bryant, *A Switch-Level Model and Simulator for MOS Digital Systems*, Technical Report 5065, Department of Computer Science, California Institute of Technology, January 1983.

[10] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[11] Xidak, Inc. *Mainsail Language Manual*, Menlo Park, CA., 1982.

[12] Winegarden, S., and D. Pannell, "Paragons for Memory Test," *1981 IEEE Test Conference*, pp. 44–48.